An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation

Victorita Dolean, Pierre Jolivet, Frédéric Nataf

An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation

Victorita Dolean Pierre Jolivet Frédéric Nataf

March 17, 2021

BEWARE: this version is out dated since December 2015
The printed version of the current version is on purchase on the SIAM website
`http://bookstore.siam.org/ot144/`
By courtesy of SIAM, the corresponding pdf file can be downloaded from Frédéric Nataf's webpage.

The purpose of this text is to offer an overview of the most popular domain decomposition methods for partial differential equations (PDE). The presentation is kept as much as possible at an elementary level with a special focus on the definitions of these methods in terms both of PDEs and of the sparse matrices arising from their discretizations. We also provide implementations written in an open source finite element software. In addition, we consider a number of methods that have not been presented in other books. We think that this book will give a new perspective and that it will complement those of Smith, Bjørstad and Gropp [25], Quarteroni and Valli [23], Mathew [20] and Toselli and Widlund[26] as well as the review article [7].

The book is addressed to computational scientists, mathematicians, physicists and, in general, to people involved in numerical simulation of partial differential equations. It can also be used as textbook for advanced undergraduate/First-Year Graduate students. The mathematical tools needed are basic linear algebra, notions of programming, variational formulation of PDEs and basic knowledge in finite element discretization.

The value of domain decomposition methods is part of a general need for parallel algorithms for professional and consumer use. We will focus on scientific computing and more specifically on the solution of the algebraic systems arising from the approximation of a partial differential equation.

Domain decomposition methods are a family of methods to solve problems of linear algebra on parallel machines in the context of simulation. In scientific computing, the first step is to model mathematically a physical phenomenon. This often leads to systems of partial differential equations such as the Navier-Stokes equations in fluid mechanics, elasticity system in solid mechanics, Schrödinger equations in quantum mechanics, Black and Scholes equation in finance, Lighthill-Witham equations for traffic, ... Functional analysis is used to study the well-posedness of the PDEs which is a necessary condition for their possible numerical approximation. Numerical analysis enables to design stable and consistant discretization schemes. This leads to discrete equations $F(u) = b \in \mathbb{R}^n$ where n is the number of degrees of freedom of the discretization. If F is linear, calculate u is a problem of linear algebra. If F is nonlinear, a method for solving is the classical Newton's method, which also leads to solving a series of linear systems.

In the past, improving performance of a program, either in speed or in the amount of data processed, was only a matter of waiting for the next generation processors. Every eighteen months, computer performance doubled. As a consequence, linear solver research would take second place to the search for new discretization schemes. But since approximately year 2005 the clock speed stagnates at 2-3 GHz. The increase in performance is almost entirely due to the increase in the number of cores per processor. All major processor vendors are producing multicore chips and now every machine

is a parallel machine. Waiting for the next generation machine does not guarantee anymore a better performance of a software. To keep doubling performance parallelism must double. It implies a huge effort in algorithmic development. Scientific computing is only one illustration of this general need in computer science. Visualization, data storage, mesh generation, operating systems, ... must be designed with parallelism in mind.

We focus here on parallel linear iterative solvers. Contrary to direct methods, the appealing feature of domain decomposition methods is that they are naturally parallel. We introduce the reader to the main classes of domain decomposition algorithms: Schwarz, Neumann-Neumann/FETI and Optimized Schwarz. For each method we start by the continuous formulation in terms of PDEs for two subdomains. We then give the definition in terms of stiffness matrices and their implementation in a free finite element package in the many subdomain case. This presentation reflects the dual nature of domain decomposition methods. They are solvers of linear systems keeping in mind that the matrices arise from the discretization of partial differential operators. As for domain decomposition methods that directly address non linearities, we refer the reader to e.g. [3] or [4] and references therein. As for iterative solvers non related to domain decomposition we refer the reader to [2] or [22] e.g. .

In Chapter 1 we start by introducing different versions of Schwarz algorithms at continuous level, having as a starting point H. Schwarz method (see [24]): Jacobi Schwarz Method (JSM), Additive Schwarz Method (ASM) and Restricted Additive Schwarz (RAS) which the default parallel solver in PETSc. The first natural feature of these algorithms are that they are equivalent to a Block-Jacobi method when the overlap is minimal. We move on to the algebraic versions of the Schwarz methods. In order to do this, several concepts are necessary: restriction and prolongation operators as well as partitions of unity which make possible the global definition. These concepts are explained in detail in the case of different type of discretizations (finite difference or finite element) and spatial dimensions. The convergence of the Schwarz method in the two-subdomain case is illustrated for one-dimensional problems and then for two-dimensional problems by using Fourier analysis.A short paragraph introduces P.L. Lions algorithm that will be considered into details in Chapter 2. The last part of the chapter is dedicated to the numerical implementation by using FreeFem++ [14] for general decompositions into subdomains.

In Chapter 2 we present Optimized Schwarz methods applied to the Helmholtz equation which models acoustic wave propagation in the frequency domain. We begin with the two subdomain case. We show the need for the use of interface conditions different from Dirichlet or Neumann boundary conditions. The Lions and Després algorithms which are based on Robin interface conditions are analyzed together with their implementa-

tions. We also show that by taking even more general interface conditions, much better convergence can be achieved at no extra cost compared to the use of Robin interface conditions. We consider the many subdomain case as well. These algorithms are the method of choice for wave propagation phenomena in the frequency regime. Such situations occur in acoustics, electromagnetics and elastodynamics.

In Chapter 3 we present the main ideas which justify the use of Krylov methods instead of stationary iterations. Since Schwarz methods introduced in Chapters 1 and 2 represent fixed point iterations applied to preconditioned global problems, and consequently not providing the fastest convergence possible, it is natural to apply Krylov methods instead. This provides the justification of using Schwarz methods as preconditioners rather than solvers. Numerical implementations and results using FreeFem++ are closing the chapter. Although some part of the presentation of some Krylov methods is not standard, readers already familiar with Krylov methods may as well skip it.

Chapter 4 is devoted to the introduction of two-level methods. In the presence of many subdomains, the performance of Schwarz algorithms, i.e. the iteration number and execution time will grow linearly with the number of subdomains in one direction. From a parallel computing point of view this translates into a lack of scalability. The latter can be achieved by adding a second level or a coarse space. This is strongly related to multigrid methods and to deflation methods from numerical linear algebra. The simplest coarse space which belongs to Nicolaides is introduced and then implemented in FreeFem++.

In Chapter 5, we show that Nicolaides coarse space (see above) is a particular case of a more general class of spectral coarse spaces which are generated by vectors issued from solving some local generalized eigenvalue problems. Then, a theory of these two-level algorithms is presented. First, a general variational setting is introduced as well as elements from the abstract theory of the two-level additive Schwarz methods (e.g. the concept of stable decomposition). The analysis of spectral and classical coarse spaces goes through some properties and functional analysis results. These results are valid for scalar elliptic PDEs. This chapter is more technical than the others and is not necessary to the sequel of the book.

Chapter 6 is devoted to the Neumann-Neumann and FETI algorithms. We start with the two subdomain case for the Poisson problem. Then, we consider the formulation in terms of stiffness matrices and stress the duality of these methods. We also establish a connection with block factorization of the stiffness matrix of the original problem. We then show that in the many subdomains case Neumann-Neumann and FETI are no longer strictly equivalent. For sake of simplicity, we give a FreeFem++ implementation of only the Neumann-Neumann algorithm. The reader is then ready to delve into the abundant litterature devoted to the use of these methods for solving

complex mechanical problems.

In Chapter 7, we return to two level methods. This time, a quite recent adaptive abstract coarse space, as well as most classical two-level methods are presented in a different light, under a common framework. Moreover, their convergence properties are proven in an abstract setting, provided that the assumptions of the Fictitious Space Lemma are satisfied. The new coarse space construction is based on solving GENeralized Eigenvalue problems in the Overlap (GenEO). The construction is provable in the sense that the condition number is given in terms of an explicit formula where the constants that appear are the maximal number of neighbors of a subdomain and a threshold prescribed by the user. The latter can be applied to a broader class of elliptic equations, which include systems of PDEs such as linear elasticity even with highly heterogeneous coefficients. From § 7.1 to § 7.6 , we give all the materials necessary to build and analyze two-level methods for Additive Schwarz methods. In section 7.7, we build a coarse space for one level Optimized Schwarz methods of Chapter 2. It is based on introducing SORAS algorithm and two related generalized eigenvalue problems. The resulting algorithm is named SORAS-GenEO-2. Section 7.8 is devoted to endow one level Neumann-Neumann algorithm of Chapter 6 with a GenEO type coarse space.

In Chapter 8 we introduce the parallel computational framework used in the parallel version of the free finite element package FreeFem++ which is currently linked with HPDDM, a C++ framework for high-performance domain decomposition methods, available at the following URL: `https://github.com/hpddm/hpddm`. For sake of simplicity we restrict ourselves to the two-level Schwarz methods. Numerical simulations of very large scale problems on high performance computers show the weak and strong scalabilities of the Schwarz methods for 2D and 3D Darcy and elasticity problems with highly heterogeneous coefficients with billions of degrees of freedom. A self contained FreeFem++ parallel script is given.

We give in Figure 1, the dependency graph of the various chapters. For instance in order to read Chapter 4 it is necessary to be familiar with both Chapters 3 *and* 1. From this graph, the reader is able to choose his way in reading the book. We suggest some possible partial readings. A reader interested in having a quick and partial view and already familiar with Krylov methods, may very well read only Chapter 1 followed by Chapter 4. For new comers to Krylov methods, reading of Chapter 3 must be intercalated between Chapter 1 and Chapter 4.
For a quick view on all Schwarz methods without entering into the technical details of coarse spaces, one could consider beginning by Chapter 1 followed by Chapter 2 and then by Chapter 3 on the use of Schwarz methods as preconditioners, to finish with Chapter 4 on classical coarse spaces.
For the more advanced reader, Chapters 5 and 7 provide the technical frame-

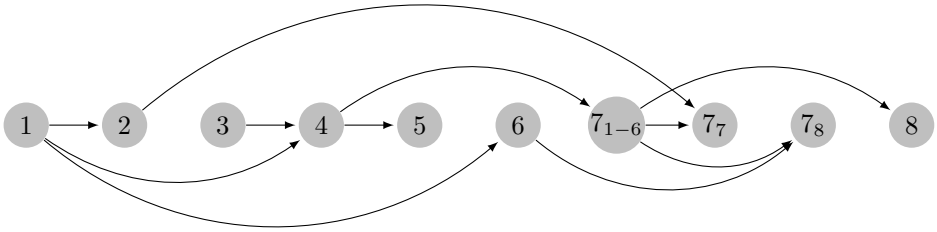

Figure 1: Dependency graph of chapters

work for the analysis and construction of more sophisticated coarse spaces. And last, but not least Chapter 8 gives the keys of parallel implementation and illustrates with large scale numerical results the previously introduced methods.

Contents

Chapter 1

Schwarz methods

1.1 Three continuous Schwarz Algorithms

Hermann Schwarz was a German analyst of the 19th century. He was interested in proving the existence and uniqueness of the Poisson problem. At his time, there were no Sobolev spaces nor Lax-Milgram theorem. The only available tool was the Fourier transform, limited by its very nature to simple geometries. In order to consider more general situations, H. Schwarz devised an iterative algorithm for solving Poisson problem set on a union of simple geometries, see [24]. For a historical presentation of these kind of methods see [13].

Let the domain Ω be the union of a disk and a rectangle, see figure 1.1. Consider the Poisson problem which consists in finding $u : \Omega \to \mathbb{R}$ such that:

$$\begin{aligned} -\Delta(u) &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{1.1}$$

Definition 1.1.1 (Original Schwarz algorithm) *The Schwarz algorithm is an iterative method based on solving alternatively sub-problems in domains*

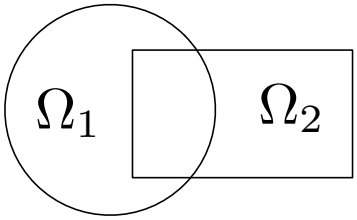

Figure 1.1: A complex domain made from the union of two simple geometries

Ω_1 *and* Ω_2. *It updates* $(u_1^n, u_2^n) \to (u_1^{n+1}, u_2^{n+1})$ *by:*

$$\begin{array}{rcll} -\Delta(u_1^{n+1}) & = & f & \text{in } \Omega_1 \\ u_1^{n+1} & = & 0 & \text{on } \partial\Omega_1 \cap \partial\Omega \quad \text{then,} \\ u_1^{n+1} & = & u_2^n & \text{on } \partial\Omega_1 \cap \overline{\Omega_2}. \end{array} \qquad \begin{array}{rcll} -\Delta(u_2^{n+1}) & = & f & \text{in } \Omega_2 \\ u_2^{n+1} & = & 0 & \text{on } \partial\Omega_2 \cap \partial\Omega \\ u_2^{n+1} & = & u_1^{n+1} & \text{on } \partial\Omega_2 \cap \overline{\Omega_1}. \end{array} \tag{1.2}$$

H. Schwarz proved the convergence of the algorithm and thus the well-posedness of the Poisson problem in complex geometries.

With the advent of digital computers, this method also acquired a practical interest as an iterative linear solver. Subsequently, parallel computers became available and a small modification of the algorithm [18] makes it suited to these architectures. Its convergence can be proved using the maximum principle [17].

Definition 1.1.2 (Parallel Schwarz algorithm) *Iterative method which solves concurrently in all subdomains,* $i = 1, 2$*:*

$$\begin{array}{rcll} -\Delta(u_i^{n+1}) & = f & \text{in} & \Omega_i \\ u_i^{n+1} & = 0 & \text{on} & \partial\Omega_i \cap \partial\Omega \\ u_i^{n+1} & = u_{3-i}^n & \text{on} & \partial\Omega_i \cap \overline{\Omega}_{3-i}. \end{array} \tag{1.3}$$

It is easy to see that if the algorithm converges, the solutions u_i^∞, $i = 1, 2$ in the intersection of the subdomains take the same values. Indeed, in the overlap $\Omega_{12} := \Omega_1 \cap \Omega_2$, let $e^\infty := u_1^\infty - u_2^\infty$. By the last line of (1.3), we know that $e^\infty = 0$ on $\partial\Omega_{12}$. By linearity of the Poisson equation, we also have that e^∞ is harmonic. Thus, e^∞ solves the homogeneous well-posed boundary value problem (BVP):

$$\begin{array}{rcll} -\Delta(e^\infty) & = 0 & \text{in} & \Omega_{12} \\ e^\infty & = 0 & \text{on} & \partial\Omega_{12} \end{array}$$

and thus $e^\infty = 0$.

Algorithms (1.2) and (1.3) act on the local functions $(u_i)_{i=1,2}$. In order to write algorithms that act on global functions we need extension operators and partitions of unity.

Definition 1.1.3 (Extension operators and partition of unity) *Let the extension operator* E_i *such that* $E_i(w_i) : \Omega \to \mathbb{R}$ *is the extension of a function* $w_i : \Omega_i \mapsto \mathbb{R}$*, by zero outside* Ω_i*. We also define the partition of unity functions* $\chi_i : \Omega_i \to \mathbb{R}$*,* $\chi_i \geq 0$ *and* $\chi_i(x) = 0$ *for* $x \in \partial\Omega_i \setminus \partial\Omega$ *and such that:*

$$w = \sum_{i=1}^{2} E_i(\chi_i \, w_{|\Omega_i}) \tag{1.4}$$

for any function $w : \Omega \mapsto \mathbb{R}$.

There are two ways to write related algorithms that act on global functions. They are given in Definitions 1.1.4 and 1.1.5.

Definition 1.1.4 (First global Schwarz iteration) *Let u^n be an approximation to the solution to the Poisson problem (1.1), u^{n+1} is computed by solving first local sub-problems:*

$$\begin{aligned} -\Delta(w_i^{n+1}) &= f \quad in \quad \Omega_i, \quad w_i^{n+1} = u^n \ on \quad \partial\Omega_i \cap \overline{\Omega}_{3-i} \\ w_i^{n+1} &= 0 \quad on \quad \partial\Omega_i \cap \partial\Omega\,. \end{aligned} \tag{1.5}$$

and then gluing them together using the partition of unity functions:

$$u^{n+1} := \sum_{i=1}^{2} E_i(\chi_i\, w_i^{n+1})\,. \tag{1.6}$$

We can prove the following property:

Lemma 1.1.1 *Algorithm* (1.5)-(1.6) *which iterates on u^n and algorithm* (1.3) *which iterates on (u_1^n, u_2^n) are equivalent.*

Proof Starting from initial guesses which satisfy $u^0 = \sum_{i=1}^{2} E_i(\chi_i\, u_i^0)$, we prove by induction that

$$u^n = \sum_{i=1}^{2} E_i(\chi_i\, u_i^n)\,. \tag{1.7}$$

holds for all $n \geq 0$. Assume the property holds at step n of the algorithm. Then, using the fact that $\chi_1 = 1$ and $\chi_2 = 0$ on $\partial\Omega_1 \cap \bar{\Omega}_2$ we have by definition that w_1^{n+1} is a solution to BVP (1.3) (with $i = 1$):

$$\begin{aligned} -\Delta(w_1^{n+1}) &= f \quad \text{in} \quad \Omega_1, \\ w_1^{n+1} &= 0 \quad \text{on} \quad \partial\Omega_1 \cap \partial\Omega, \\ w_1^{n+1} = u^n = \sum_{i=1}^{2} E_i(\chi_i\, u_i^n) &= u_2^n \quad \text{on} \quad \partial\Omega_1 \cap \overline{\Omega}_2. \end{aligned} \tag{1.8}$$

and thus $w_1^{n+1} = u_1^{n+1}$. The proof is the same for $w_2^{n+1} = u_2^{n+1}$. Finally, we have using (1.6):

$$u^{n+1} = \sum_{i=1}^{2} E_i(\chi_i\, w_i^n) = \sum_{i=1}^{2} E_i(\chi_i\, u_i^n)\,.$$

■

This result can be seen as a continuous version of the algebraic formulation established in [12].

We introduce in Algorithm 1 another formulation to algorithm (1.5)-(1.6) in terms of the continuous residual $r^n := f + \Delta u^n$. This way, we get closer to the algebraic definition of domain decomposition methods. Algorithm 1 is named RAS which stands for Restricted Additive Schwarz.

Lemma 1.1.2 (Equivalence between Schwarz' algorithm and RAS) *The algorithm defined by* (1.12), (1.13) *and* (1.14) *is called the continuous RAS algorithm. It is equivalent to the Schwarz' algorithm* (1.3).

Proof Here, we have to prove the equality

$$u^n = E_1(\chi_1 u_1^n) + E_2(\chi_2 u_2^n)\,,$$

where $u_{1,2}^n$ is given by (1.3) and u^n is given by (1.12)-(1.13)-(1.14). We assume that the property holds for the initial guesses:

$$u^0 = E_1(\chi_1 u_1^0) + E_2(\chi_2 u_2^0)$$

and proceed by induction assuming the property holds at step n of the algorithm, i.e. $u^n = E_1(\chi_1 u_1^n) + E_2(\chi_2 u_2^n)$. From (1.14) we have

$$u^{n+1} = E_1(\chi_1(u^n + v_1^n)) + E_2(\chi_2(u^n + v_2^n))\,. \tag{1.9}$$

We prove now that $u_{|\Omega_1}^n + v_1^n = u_1^{n+1}$ by proving that $u_{|\Omega_1}^n + v_1^n$ satisfies (1.3) as u_1^{n+1} does. We first note that, using (1.13)-(1.12) we have:

$$\begin{aligned} &-\Delta(u^n + v_1^n) = -\Delta(u^n) + r^n = -\Delta(u^n) + f + \Delta(u^n) = f \quad \text{in } \Omega_1, \\ &u^n + v_1^n = u^n \quad \text{on } \partial\Omega_1 \cap \overline{\Omega_2}, \end{aligned} \tag{1.10}$$

It remains to prove that

$$u^n = u_2^n \text{ on } \partial\Omega_1 \cap \overline{\Omega_2}\,.$$

By the induction hypothesis we have $u^n = E_1(\chi_1 u_1^n) + E_2(\chi_2 u_2^n)$. On $\partial\Omega_1 \cap \overline{\Omega_2}$, we have $\chi_1 \equiv 0$ and thus $\chi_2 \equiv 1$. So that on $\partial\Omega_1 \cap \overline{\Omega_2}$ we have :

$$u^n = \chi_1 u_1^n + \chi_2 u_2^n = u_2^n\,. \tag{1.11}$$

Finally from (1.10) and (1.11) we can conclude that $u_{|\Omega_1}^n + v_1^n$ satisfies problem (1.3) and is thus equal to u_1^{n+1}. The same holds for domain Ω_2, $u_{|\Omega_2}^n + v_2^n = u_2^{n+1}$. Then equation (1.9) reads

$$u^{n+1} = E_1(\chi_1 u_1^{n+1}) + E_2(\chi_2 u_2^{n+1})$$

which ends the proof of the equivalence between Schwarz' algorithm and the continuous RAS algorithm (1.12)–(1.14). ■

Another global variant of the parallel Schwarz algorithm (1.3) consists in replacing formula (1.6) by a simpler formula not based on the partition of unity.

Algorithm 1 RAS algorithm at the continuous level

1. Compute the residual $r^n : \Omega \to \mathbb{R}$:

$$r^n := f + \Delta(u^n) \tag{1.12}$$

2. For $i = 1, 2$ solve for a local correction v_i^n:

$$-\Delta(v_i^n) \quad = \quad r^n \quad \text{in } \Omega_i\,, \quad v_i^n = 0 \text{ on } \quad \partial\Omega_i \tag{1.13}$$

3. Compute an average of the local corrections and update u^n:

$$u^{n+1} = u^n + E_1(\chi_1 v_1^n) + E_2(\chi_2 v_2^n)\,. \tag{1.14}$$

where $(\chi_i)_{i=1,2}$ and $(E_i)_{i=1,2}$ define a partition of unity as in defined in section 1.1 equation (1.4).

Definition 1.1.5 (Second global Schwarz iteration) *Let u^n be an approximation to the solution to the Poisson problem (1.1), u^{n+1} is computed by solving first local sub-problems* (1.5) *and then gluing them together without the use of the partition of unity functions:*

$$u^{n+1} := \sum_{i=1}^{2} E_i(w_i^{n+1})\,. \tag{1.15}$$

It is easy to check that this algorithm is equivalent to Algorithm 2 which is called ASM (Additive Schwarz method).

Algorithm 2 ASM algorithm at the continuous level

1. Compute the residual $r^n : \Omega \to \mathbb{R}$:

$$r^n := f + \Delta(u^n) \tag{1.16}$$

2. For $i = 1, 2$ solve for a local correction v_i^n:

$$-\Delta(v_i^n) \quad = \quad r^n \text{ in } \Omega_i\,, \quad v_i^n = 0 \text{ on } \partial\Omega_i \tag{1.17}$$

3. Update u^n:

$$u^{n+1} = u^n + E_1(v_1^n) + E_2(v_2^n)\,. \tag{1.18}$$

To sum up, starting from the original Schwarz algorithm (1.2) that is sequential, we have thus three continuous algorithms that are essentially parallel:

- Algorithm (1.3) Jacobi Schwarz Method (JSM)
- Algorithm (1.12)-(1.13)-(1.14) Restricted Additive Schwarz (RAS)
- Algorithm (1.16)-(1.17)-(1.18) Additive Schwarz Method (ASM)

The discrete version of the first algorithm is seldom implemented since it involves duplication of unknowns. The discrete version of the second algorithm is the *restricted additive Schwarz* method (RAS, see[5, 6]) which is the default parallel solver in the package PETSC [1]. The discrete version of the third algorithm is the *additive Schwarz method* (ASM) for which many theoretical results have been derived, see [26] and references therein. The latter term was introduced first by Dryja and Widlund in [11] for a variant of the algorithm firstly introduced at continuous level in [21].

1.2 Connection with the Block Jacobi algorithm

In the previous section we have noticed that the three methods illustrate different points of view of the Schwarz iteration, the continuous aspect emphasized the interest of the overlap (see § 1.5), which is hidden in the discrete formulation. When going to the discrete level, we will see that Schwarz algorithm is, from a linear algebra point of view, a variation of a block-Jacobi algorithm.

We first recall the definition of a block Jacobi algorithm and then establish a connection with the Schwarz algorithms. Let us consider a linear system:

$$A\mathbf{U} = \mathbf{F} \tag{1.19}$$

with a matrix A of size $m \times m$, a right-hand side $\mathbf{F} \in \mathbb{R}^m$ and a solution $\mathbf{U} \in \mathbb{R}^m$ where m is an integer. The set of indices $\{1, \ldots, m\}$ is partitioned into two sets

$$\mathcal{N}_1 := \{1, \ldots, m_s\} \text{ and } \mathcal{N}_2 := \{m_s + 1, \ldots, m\}.$$

Let $\mathbf{U}_1 := (U_k)_{k \in \mathcal{N}_1} := \mathbf{U}_{|\mathcal{N}_1}$, $\mathbf{U}_2 := (U_k)_{k \in \mathcal{N}_2} := \mathbf{U}_{|\mathcal{N}_2}$ and similarly $\mathbf{F}_1 := \mathbf{F}_{|\mathcal{N}_1}$, $F_2 := F_{|\mathcal{N}_2}$.
The linear system has the following block form:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \mathbf{U}_1 \\ \mathbf{U}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{F}_1 \\ \mathbf{F}_2 \end{pmatrix}$$

where $A_{ij} := A_{|\mathcal{N}_i \times \mathcal{N}_j}$, $1 \le i, j \le 2$.

Definition 1.2.1 (Jacobi algorithm) *Let D be the diagonal of A, the Jacobi algorithm reads:*

$$D\mathbf{U}^{n+1} = D\mathbf{U}^n + (\mathbf{F} - A\mathbf{U}^n),$$

or equivalently,

$$\mathbf{U}^{n+1} = \mathbf{U}^n + D^{-1}(\mathbf{F} - A\mathbf{U}^n) = \mathbf{U}^n + D^{-1}\mathbf{r}^n\,,$$

where $\mathbf{r}^n = F - A\mathbf{U}^n$ *is the residual of the equation.*

We now define a block Jacobi algorithm.

Definition 1.2.2 (Block-Jacobi algorithm) *The block-Jacobi algorithm reads:*

$$\begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}\begin{pmatrix} \mathbf{U}_1^{n+1} \\ \mathbf{U}_2^{n+1} \end{pmatrix} = \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}\begin{pmatrix} \mathbf{U}_1^{n} \\ \mathbf{U}_2^{n} \end{pmatrix} + \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} - A\begin{pmatrix} \mathbf{U}_1^{n} \\ \mathbf{U}_2^{n} \end{pmatrix} \tag{1.20}$$

or equivalently

$$\begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}\begin{pmatrix} \mathbf{U}_1^{n+1} \\ \mathbf{U}_2^{n+1} \end{pmatrix} = \begin{pmatrix} F_1 - A_{12}\,\mathbf{U}_2^{n} \\ F_2 - A_{21}\,\mathbf{U}_1^{n} \end{pmatrix}. \tag{1.21}$$

In order to have a more compact form of the previous algorithm, we introduce R_1 the restriction operator from $\mathcal{N}$ into $\mathcal{N}_1$ and similarly R_2 the restriction operator from $\mathcal{N}$ into $\mathcal{N}_2$. The transpose operator R_i^T are extensions operators from $\mathcal{N}_i$ into $\mathcal{N}$. Note that $A_{ii} = R_i A R_i^T$.

Lemma 1.2.1 (Compact form of a block-Jacobi algorithm) *The algorithm (1.21) can be re-written as*

$$\boxed{\mathbf{U}^{n+1} = \mathbf{U}^n + \left(R_1^T(R_1AR_1^T)^{-1}R_1 + R_2^T(R_2AR_2^T)^{-1}R_2\right)\mathbf{r}^n\,.} \tag{1.22}$$

Proof Let $\mathbf{U}^n = (\mathbf{U}_1^{nT}, \mathbf{U}_2^{nT})^T$, algorithm (1.21) becomes

$$\begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}\mathbf{U}^{n+1} = F - \begin{pmatrix} 0 & A_{12} \\ A_{21} & 0 \end{pmatrix}\mathbf{U}^{n}\,. \tag{1.23}$$

On the other hand, equation (1.20) can be rewritten equivalently

$$\begin{pmatrix} \mathbf{U}_1^{n+1} \\ \mathbf{U}_2^{n+1} \end{pmatrix} = \begin{pmatrix} \mathbf{U}_1^{n} \\ \mathbf{U}_2^{n} \end{pmatrix} + \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}^{-1}\begin{pmatrix} \mathbf{r}_1^{n} \\ \mathbf{r}_2^{n} \end{pmatrix} \Leftrightarrow \mathbf{U}^{n+1} = \mathbf{U}^n + \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & A_{22}^{-1} \end{pmatrix}\mathbf{r}^n \tag{1.24}$$

where $\mathbf{r}_i^n := \mathbf{r}^n_{|\mathcal{N}_i}$, $i = 1, 2$. By taking into account that

$$\begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & 0 \end{pmatrix} = R_1^T A_{11}^{-1} R_1 = R_1^T (R_1 A R_1^T)^{-1} R_1$$

and

$$\begin{pmatrix} 0 & 0 \\ 0 & A_{22}^{-1} \end{pmatrix} = R_2^T A_{22}^{-1} R_2 = R_2^T (R_2 A R_2^T)^{-1} R_2\,,$$

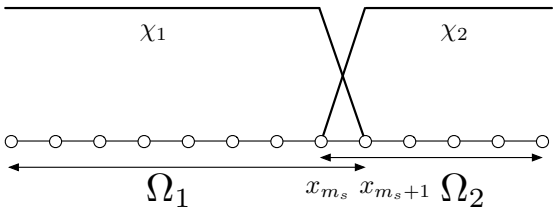

Figure 1.2: Domain decomposition with minimal overlap and partition of unity

the conclusion follows easily. ■

In order to establish a connection with the Schwarz algorithms, consider the following BVP on $\Omega := (0,1)$: Find u such that

$$\begin{array}{rcl} -\Delta u &=& f, \quad \text{in} \quad \Omega \\ u(0) &=& u(1) = 0\,. \end{array}$$

We discretize it by a three point finite difference scheme on the grid $x_j := j\,h$, $1 \le j \le m$ where $h := 1/(m+1)$. Let $u_j \simeq u(x_j)$, $f_j := f(x_j)$, $1 \le j \le m$ and $\mathbf{U} = (u_j)_{1\le j\le m}$, $\mathbf{F} = (f_j)_{1\le j\le m}$ satisfy equation (1.19) where A is the tridiagonal matrix $A_{j\,j} := 2/h^2$ and $A_{j\,j+1} = A_{j+1\,j} := -1/h^2$.

Let domains $\Omega_1 := (0,(m_s+1)\,h)$ and $\Omega_2 := (m_s\,h,1)$ define an overlapping decomposition with a minimal overlap of width h. The discretization of (1.5) for domain Ω_1 reads

$$\left\{ \begin{array}{rcl} -\dfrac{u^{n+1}_{1,j-1} - 2u^{n+1}_{1,j} + u^{n+1}_{1,j+1}}{h^2} &=& f_j,\ 1 \le j \le m_s \\ u^{n+1}_{1,0} &=& 0 \\ u^{n+1}_{1,m_s+1} &=& u^{n}_{2,m_s+1} \end{array} \right. .$$

Solving for $\mathbf{U}_1^{n+1} = (u_{1,j}^{n+1})_{1\le j\le m_s}$ corresponds to solving a Dirichlet boundary value problem in subdomain Ω_1 with Dirichlet data taken from the other subdomain at the previous step. With the notations introduced previously, $\mathbf{U}_1^{n+1}$ satisfies

$$A_{11}\mathbf{U}_1^{n+1} + A_{12}\mathbf{U}_2^{n} = \mathbf{F}_1\,.$$

Similarly, we have

$$A_{22}\mathbf{U}_2^{n+1} + A_{21}\mathbf{U}_1^{n} = \mathbf{F}_2\,.$$

These two equations are equivalent to (1.21) and represent the discretization of the JSM method (1.3).

The discrete counterpart of the extension operator E_1 (resp. E_2) is defined by $E_1(\mathbf{U}_1) = (\mathbf{U}_1^T, 0)^T$ (resp. $E_2(\mathbf{U}_2) = (0, \mathbf{U}_2^T)^T$). The discretization of the ASM (1.15) is then given by equation (1.23). *Thus when the overlap is minimal, the ASM method reduces to the block Jacobi algorithm.*

Let χ_i, $i = 1, 2$ be the piecewise linear functions that define a partition of unity on the domain decomposition, see Figure 1.2. In this very simple configuration,

$$\chi_1(x) = \begin{cases} 1 & \text{if } 0 \le x \le x_{m_s} \\ \dfrac{x_{m_s+1} - x}{h} & \text{if } x_{m_s} \le x \le x_{m_s+1} \end{cases}$$

and

$$\chi_2(x) = \begin{cases} \dfrac{x - x_{m_s}}{h} & \text{if } x_{m_s} \le x \le x_{m_s+1} \\ 1 & \text{if } x_{m_s+1} \le x \le 1 \end{cases} .$$

Functions χ_i, $i = 1, 2$ define a partition of unity in the sense of (1.4). Since the overlap is minimal, the discretization of (1.6) is equivalent to that of (1.15). Thus RAS reduces, in this case, to ASM.

Remark 1.2.1 *In conclusion when the overlap is minimal the discrete counterparts of the three Schwarz methods of section 1.1 are equivalent to the same block Jacobi algorithm. Notice here a counter-intuitive feature: a non overlapping decomposition of the set of indices $\mathcal{N}$ corresponds to a geometric decomposition of the domain Ω with minimal overlap.*

1.3 Algebraic algorithms: discrete partition of unity

Our goal is to introduce in the general case the algebraic counterparts of algorithms RAS and ASM defined in § 1.1. The simplest way to do so is to write the iterative method in terms of residuals as is done in equation (1.22). In order to do this, we need to settle some elements necessary in this writing. One of them is the proper definition of the partition of unity.

At the continuous level (partial differential equations), the main ingredients of the partition of unity are

- An open domain Ω and an overlapping decomposition into N open subsets $\Omega = \cup_{i=1}^{N} \Omega_i$.
- A function $u : \Omega \to \mathbb{R}$.
- The extension operator E_i of a function $\Omega_i \to \mathbb{R}$ to a function $\Omega \to \mathbb{R}$ equals to zero in $\Omega \backslash \Omega_i$.
- The partition of unity functions χ_i, $1 \le i \le N$ introduced in formula (1.4) which verify for all functions $u : \Omega \to \mathbb{R}$:

$$u = \sum_{i=1}^{2} E_i(\chi_i \, u_{|\Omega_i}).$$

We can give a similar definition at the discrete level.

Definition 1.3.1 (Algebraic partition of unity) *At the discrete level, the main ingredients of the partition of unity are*

- *A set indices of degrees of freedom $\mathcal{N}$ and a decomposition into N subsets $\mathcal{N} = \cup_{i=1}^{N} \mathcal{N}_i$.*
- *A vector $\mathbf{U} \in \mathbb{R}^{\#\mathcal{N}}$.*
- *The restriction of a vector $\mathbf{U} \in \mathbb{R}^{\#\mathcal{N}}$ to a subdomain Ω_i, $1 \leq i \leq N$ can be expressed as $R_i \mathbf{U}$ where R_i is a rectangular $\#\mathcal{N}_i \times \#\mathcal{N}$ Boolean matrix. The extension operator will be the transpose matrix R_i^T.*
- *The partition of unity "functions" at discrete level correspond to diagonal matrices of size $\#\mathcal{N}_i \times \#\mathcal{N}_i$ with non negative entries such that for all vectors $\mathbf{U} \in \mathbb{R}^{\#\mathcal{N}}$*

$$U = \sum_{i=1}^{N} R_i^T D_i R_i U ,$$

or in other words

$$\boxed{I_d = \sum_{i=1}^{N} R_i^T D_i R_i} \tag{1.25}$$

where $I_d \in \mathbb{R}^{\#\mathcal{N} \times \#\mathcal{N}}$ is the identity matrix.

As pointed out in Remark 1.2.1 an overlapping decomposition of a domain Ω might correspond to a partition of the set of indices.

In the following we will give some simple examples where all the ingredients of the Definition 1.3.1 are detailed and we will check that (1.25) is verified in those cases.

1.3.1 Two subdomain case in one dimension

1d Algebraic setting

We start from the 1d example of § 1.2 with $n = 5$, $n_s = 3$ so that the set of indices $\mathcal{N} := \{1, \ldots, 5\}$ is partitioned into two sets, see Figure 1.3

$$\mathcal{N}_1 := \{1, 2, 3\} \text{ and } \mathcal{N}_2 := \{4, 5\} .$$

Then, matrix R_1 is of size 3×5 and matrix R_2 is of size 2×5:

$$R_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \text{ and } R_2 = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} ,$$

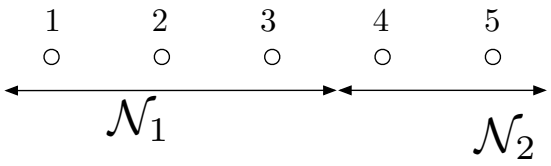

Figure 1.3: Algebraic partition of the set of indices

and

$$R_1^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ and } R_2^T = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

We also have

$$D_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } D_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

It is clear that relation (1.25) holds.

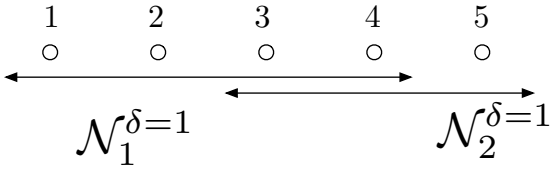

Figure 1.4: Algebraic decomposition of the set of indices into overlapping subsets

Consider now the case where each subset is extended with a neighboring point, see Figure 1.4:

$$\mathcal{N}_1^{\delta=1} := \{1, 2, 3, 4\} \text{ and } \mathcal{N}_2^{\delta=1} := \{3, 4, 5\}.$$

Then, matrices R_1 and R_2 are:

$$R_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \text{ and } R_2 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

The simplest choices for the partition of unity matrices are

$$D_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ and } D_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

or

$$D_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 \end{pmatrix} \text{ and } D_2 = \begin{pmatrix} 1/2 & 0 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Again, it is clear that relation (1.25) holds.

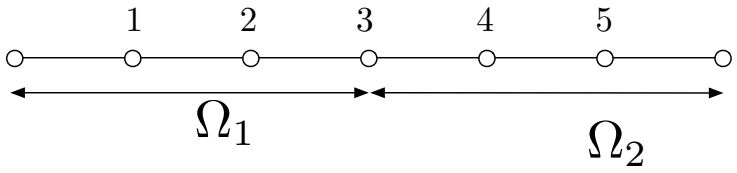

Figure 1.5: Finite element partition of the mesh

1d Finite element decomposition

We still consider the 1d example with a decomposition into two subdomains but now in a finite element spirit. A *partition* of the 1D mesh of Figure 1.5 corresponds to an *overlapping* decomposition of the set of indices:

$$\mathcal{N}_1 := \{1, 2, 3\} \text{ and } \mathcal{N}_2 := \{3, 4, 5\}.$$

Then, matrices R_1 and R_2 are:

$$R_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \text{ and } R_2 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

In order to satisfy relation (1.25), the simplest choice for the partition of unity matrices is

$$D_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1/2 \end{pmatrix} \text{ and } D_2 = \begin{pmatrix} 1/2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Consider now the situation where we add a mesh to each subdomain, see Figure 1.6. Accordingly, the set of indices is decomposed as:

$$\mathcal{N}_1^{\delta=1} := \{1, 2, 3, 4\} \text{ and } \mathcal{N}_2^{\delta=1} := \{2, 3, 4, 5\}.$$

Then, matrices R_1 and R_2 are:

$$R_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \text{ and } R_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

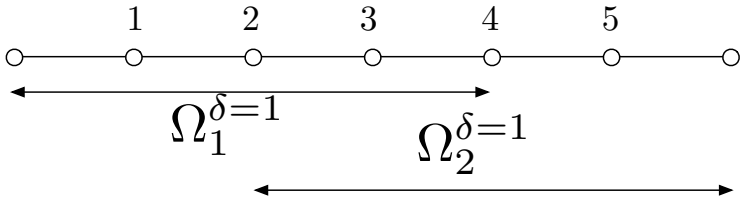

Figure 1.6: Finite element decomposition of the mesh into overlapping subdomains

In order to satisfy relation (1.25), the simplest choice for the partition of unity matrices is

$$D_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ and } D_2 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Another possible choice that will satisfy relation (1.25) as well is

$$D_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 \end{pmatrix} \text{ and } D_2 = \begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

1.3.2 Multi dimensional problems and many subdomains

In the general case, the set of indices $\mathcal{N}$ can be partitioned by an automatic graph partitioner such as METIS[15] or SCOTCH [8]. From the input matrix A, a connectivity graph is created. Two indices $i, j \in \mathcal{N}$ are connected if the matrix coefficient $A_{ij} \neq 0$. Usually, even if matrix A is not symmetric, the connectivity graph is symmetrized. Then algorithms that find a good partitioning of the vertices even for highly unstructured graphs are used. This distribution must be done so that the number of elements assigned to each processor is roughly the same, and the number of adjacent elements assigned to different processors is minimized (graph cuts). The goal of the first condition is to balance the computations among the processors. The goal of the second condition is to minimize the communication resulting from the placement of adjacent elements to different processors.

Multi-D algebraic setting

Let us consider a partition into N subsets (see Figure 1.7):

$$\mathcal{N} := \bigcup_{i=1}^{N} \mathcal{N}_i, \quad \mathcal{N}_i \cap \mathcal{N}_j = \varnothing \text{ for } i \neq j\,. \tag{1.26}$$

Let R_i be the restriction matrix from set $\mathcal{N}$ to the subset $\mathcal{N}_i$ and D_i the identity matrix of size $\#\mathcal{N}_i \times \#\mathcal{N}_i$, $1 \le i \le N$. Then, relation (1.25) is satisfied.

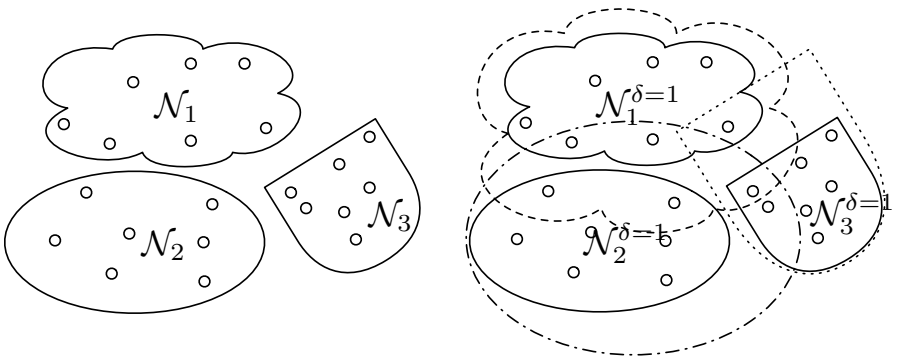

Figure 1.7: Partition and overlapping decomposition of the set of indices

Consider now the case where each subset $\mathcal{N}_i$ is extended with its direct neighbors to form $\mathcal{N}_i^{\delta=1}$, see Figure 1.7. Let R_i be the restriction matrix from set $\mathcal{N}$ to the subset $\mathcal{N}_i^{\delta=1}$ and D_i be a diagonal matrix of size $\#\mathcal{N}_i^{\delta=1} \times \#\mathcal{N}_i^{\delta=1}$, $1 \le i \le N$. For the choice of the coefficients of D_i there are two main options. The simplest one is to define it as a Boolean matrix:

$$(D_i)_{jj} := \begin{cases} 1 \text{ if } j \in \mathcal{N}_i, \\ 0 \text{ if } j \in \mathcal{N}_i^{\delta=1} \backslash \mathcal{N}_i. \end{cases}$$

Then, relation (1.25) is satisfied. Another option is to introduce for all $j \in \mathcal{N}$ the set of subsets having j as an element:

$$\mathcal{M}_j := \{1 \le i \le N | \ j \in \mathcal{N}_i^{\delta=1}\}.$$

Then, define

$$(D_i)_{jj} := 1/\#\mathcal{M}_j, \text{ for } j \in \mathcal{N}_i^{\delta=1}.$$

Then, relation (1.25) is satisfied.

Multi-D finite element decomposition

Partitioning a set of indices is well adapted to an algebraic framework. In a finite element setting, the computational domain is the union of elements of the finite element mesh $\mathcal{T}_h$. A geometric partition of the computational domain is natural. Here again, graph partitioning can be used by first modeling the finite element mesh by a graph, and then partitioning the elements into N parts $(\mathcal{T}_{i,h})_{1 \le i \le N}$, see Figure 1.8. By adding to each part

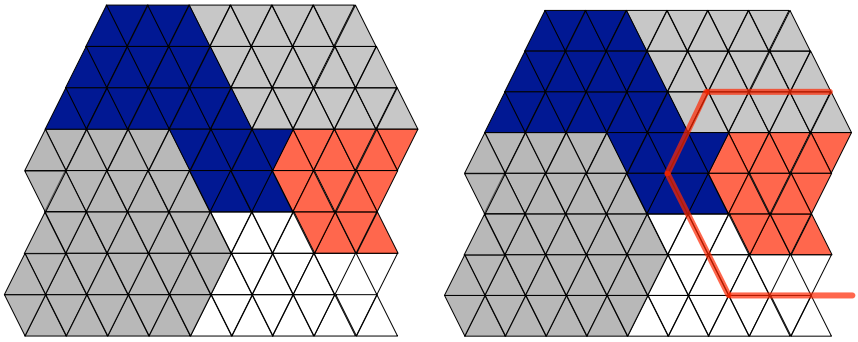

Figure 1.8: Left: Finite element partition; Right: one layer extension of the right subdomain

layers of elements, it is possible to create overlapping subdomains resolved by the finite element meshes:

$$\Omega_i = \bigcup_{\tau \in \mathcal{T}_{i,h}} \tau \quad \text{for } 1 \le i \le N\,. \tag{1.27}$$

Let $\{\phi_k\}_{k\in\mathcal{N}}$ be a basis of the finite element space. We define

$$\mathcal{N}_i := \{k \in \mathcal{N} : \operatorname{supp}(\phi_k) \cap \Omega_i \neq \varnothing\}\, 1 \le i \le N. \tag{1.28}$$

For each degree of freedom $k \in \mathcal{N}$, let

$$\mu_k := \#\{j : 1 \le j \le N \text{ and } \operatorname{supp}(\phi_k) \cap \Omega_j \neq \varnothing\}\,.$$

Let R_i be the restriction matrix from set $\mathcal{N}$ to the subset $\mathcal{N}_i$ and D_i be a diagonal matrix of size $\#\mathcal{N}_i \times \#\mathcal{N}_i$, $1 \le i \le N$ such that

$$(D_i)_{kk} := 1/\mu_k,\ k \in \mathcal{N}_i.$$

Then, relation (1.25) is satisfied.

1.4 Iterative Schwarz methods: RAS, ASM

In a similar way to what was done for the block Jacobi algorithm in equation (1.22), we can define RAS (the counterpart of Algorithm (1.12)–(1.14)) and ASM algorithms (the counterpart of Algorithm (1.16)-(1.18)).

Definition 1.4.1 (RAS algorithm) *The iterative RAS algorithm is the preconditioned fixed point iteration defined by*

$$\mathbf{U}^{n+1} = \mathbf{U}^n + M_{RAS}^{-1}\mathbf{r}^n,\ \mathbf{r}^n := \mathbf{F} - A\,\mathbf{U}^n$$

where the matrix

$$\boxed{M_{RAS}^{-1} := \sum_{i=1}^{N} R_i^T D_i \left(R_i A R_i^T\right)^{-1} R_i} \tag{1.29}$$

is called the RAS preconditioner.

Definition 1.4.2 (ASM algorithm) *The iterative ASM algorithm is the preconditioned fixed point iteration defined by*

$$\mathbf{U}^{n+1} = \mathbf{U}^n + M_{ASM}^{-1}\mathbf{r}^n,\ \mathbf{r}^n := \mathbf{F} - A\,\mathbf{U}^n$$

where the matrix

$$\boxed{M_{ASM}^{-1} := \sum_{i=1}^{N} R_i^T \left(R_i A R_i^T\right)^{-1} R_i} \tag{1.30}$$

is called the ASM preconditioner.

1.5 Convergence analysis

In order to have an idea about the convergence of these methods, we perform a simple yet revealing analysis. We consider in § 1.5.1. a one dimensional domain decomposed into two subdomains. This shows that the size of the overlap between the subdomains is key to the convergence of the method. In § 1.5.2 an analysis in the multi dimensional case is carried out by a Fourier analysis. It reveals that the high frequency component of the error is very quickly damped thanks to the overlap whereas the low frequency part will demand a special treatment, see chapter 4 on coarse spaces and two-level methods.

1.5.1 1d case: a geometrical analysis

In the 1D case, the original sequential Schwarz method (1.2) can be analyzed easily. Let $L > 0$ and the domain $\Omega = (0, L)$ be decomposed into two subodmains $\Omega_1 := (0, L_1)$ and $\Omega_2 := (l_2, L)$ with $l_2 \leq L_1$. By linearity of the equation and of the algorithm the error $e_i^n := u_i^n - u_{|\Omega_i}$, $i = 1, 2$ satisfies

$$\begin{array}{rcll} -\dfrac{d^2 e_1^{n+1}}{dx^2} & = & 0 & \text{in } (0, L_1) \\ e_1^{n+1}(0) & = & 0 & \\ e_1^{n+1}(L_1) & = & e_2^n(L_1) & \end{array} \quad \text{then,} \quad \begin{array}{rcll} -\dfrac{d^2 e_2^{n+1}}{dx^2} & = & 0 & \text{in } (l_2, L) \\ e_2^{n+1}(l_2) & = & e_1^{n+1}(l_2) & \\ e_2^{n+1}(L) & = & 0\,. & \end{array} \tag{1.31}$$

Thus the errors are affine functions in each subdomain:

$$e_1^{n+1}(x) = e_2^n(L_1)\,\frac{x}{L_1} \text{ and } e_2^{n+1}(x) = e_1^{n+1}(l_2)\,\frac{L - x}{L - l_2}\,.$$

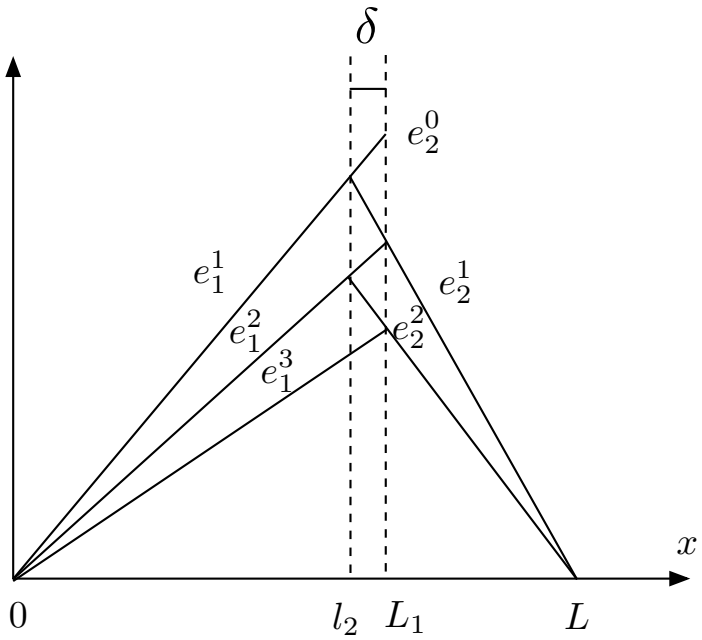

Figure 1.9: Convergence of the Schwarz method

Thus, we have

$$e_2^{n+1}(L_1) = e_1^{n+1}(l_2)\,\frac{L-L_1}{L-l_2} = e_2^n(L_1)\,\frac{l_2}{L_1}\,\frac{L-L_1}{L-l_2}\,.$$

Let $\delta := L_1 - l_2$ denote the size of the overlap, we have

$$e_2^{n+1}(L_1) = \frac{l_2}{l_2+\delta}\,\frac{L-l_2-\delta}{L-l_2}\,e_2^n(L_1) = \frac{1-\delta/(L-l_2)}{1+\delta/l_2}\,e_2^n(L_1)\,.$$

We see that the following quantity is the convergence factor of the algorithm

$$\boxed{\rho_1 = \frac{1-\delta/(L-l_2)}{1+\delta/l_2}}$$

It is clear that $\delta > 0$ is a sufficient and necessary condition to have convergence. The convergence becomes faster as the ratio of the size of the overlap over the size of a subdomain is bigger. A geometric illustration of the history of the convergence can be found in figure 1.9.

1.5.2 2d case: Fourier analysis for two subdomains

For sake of simplicity we consider the plane $\mathbb{R}^2$ decomposed into two half-planes $\Omega_1 = (-\infty, \delta) \times \mathbb{R}$ and $\Omega_2 = (0, \infty) \times \mathbb{R}$ with an overlap of size $\delta > 0$.

We choose as an example a symmetric positive definite problem ($\eta > 0$)

$$\begin{aligned}(\eta - \Delta)(u) = f \quad \text{in} \quad \mathbb{R}^2,\\ u \text{ is bounded at infinity},\end{aligned}$$

The Jacobi-Schwarz method for this problem is the following iteration

$$\begin{aligned}(\eta - \Delta)(u_1^{n+1}) &= f(x,y), \quad (x,y) \in \Omega_1\\ u_1^{n+1}(\delta, y) &= u_2^n(\delta, y), \quad y \in \mathbb{R}\end{aligned} \tag{1.32}$$

and

$$\begin{aligned}(\eta - \Delta)(u_2^{n+1}) &= f(x,y), \quad (x,y) \in \Omega_2\\ u_2^{n+1}(0, y) &= u_1^n(0, y), \quad y \in \mathbb{R}\end{aligned} \tag{1.33}$$

with the local solutions u_j^{n+1}, $j = 1, 2$ bounded at infinity.
In order to compute the convergence factor, we introduce the errors

$$e_i^n := u_i^n - u|_{\Omega_i},\ i = 1, 2.$$

By linearity, the errors satisfy the above algorithm with $f = 0$:

$$\begin{aligned}(\eta - \Delta)(e_1^{n+1}) &= 0, \quad (x,y) \in \Omega_1\\ e_1^{n+1}(\delta, y) &= e_2^n(\delta, y), \quad y \in \mathbb{R}\end{aligned} \tag{1.34}$$

and

$$\begin{aligned}(\eta - \Delta)(e_2^{n+1}) &= 0, \quad (x,y) \in \Omega_2\\ e_2^{n+1}(0, y) &= e_1^n(0, y), \quad y \in \mathbb{R}\end{aligned} \tag{1.35}$$

with e_j^{n+1} bounded at infinity.

By taking the partial Fourier transform of the first line of (1.34) in the y direction we get:

$$\left(\eta - \frac{\partial^2}{\partial x^2} + k^2\right)(\hat{e}_1^{n+1}(x,k)) = 0 \quad \text{in} \quad \Omega_1.$$

For a given Fourier variable k, this is an ODE whose solution is sought in the form

$$\hat{e}_1^{n+1}(x,k) = \sum_j \gamma_j(k) \exp(\lambda_j(k)x).$$

A simple computation gives

$$\lambda_1(k) = \lambda^+(k),\ \lambda_2(k) = \lambda^-(k), \text{ with } \lambda^\pm(k) = \pm\sqrt{\eta + k^2}.$$

Therefore we have

$$\hat{e}_1^{n+1}(x,k) = \gamma_+^{n+1}(k) \exp(\lambda^+(k)x) + \gamma_-^{n+1}(k) \exp(\lambda^-(k)x).$$

Since the solution must be bounded at $x = -\infty$, this implies that $\gamma_-^{n+1}(k) \equiv 0$. Thus we have

$$\hat{e}_1^{n+1}(x,k) = \gamma_+^{n+1}(k)\exp(\lambda^+(k)x)$$

or equivalently, by changing the value of the coefficient γ_+,

$$\hat{e}_1^{n+1}(x,k) = \gamma_1^{n+1}(k)\exp(\lambda^+(k)(x-\delta))$$

and similarly, in domain Ω_2 we have:

$$\hat{e}_2^{n+1}(x,k) = \gamma_2^{n+1}(k)\exp(\lambda^-(k)x)$$

with $\gamma_{1,2}^{n+1}$ to be determined. From the interface conditions we get

$$\gamma_1^{n+1}(k) = \gamma_2^n(k)\ \exp(\lambda^-(k)\delta)$$

and

$$\gamma_2^{n+1}(k) = \gamma_1^n(k)\exp(-\lambda^+(k)\delta).$$

Combining these two and denoting $\lambda(k) = \lambda^+(k) = -\lambda^-(k)$, we get for $i = 1, 2$,

$$\gamma_i^{n+1}(k) = \rho(k;\alpha,\delta)^2\,\gamma_i^{n-1}(k)$$

with ρ the convergence factor given by:

$$\boxed{\rho(k;\alpha,\delta) = \exp(-\lambda(k)\delta),\ \lambda(k) = \sqrt{\eta + k^2}.} \tag{1.36}$$

A graphical representation can be found in Figure 1.10 for some values of the overlap. This formula deserves a few remarks.

Remark 1.5.1 *We have the following properties:*

- *For all $k \in \mathbb{R}$, $\rho(k) < \exp(-\sqrt{\eta}\,\delta) < 1$ so that $\gamma_i^n(k) \to 0$ uniformly as n goes to infinity.*
- *$\rho \to 0$ as k tends to infinity, high frequency modes of the error converge very fast.*
- *When there is no overlap ($\delta = 0$), $\rho = 1$ and there is stagnation of the method.*

1.6 More sophisticated Schwarz methods: P.L. Lions' Algorithm

During the last decades, more sophisticated Schwarz methods were designed, namely the optimized Schwarz methods. These methods are based on a classical domain decomposition, but they use more effective transmission

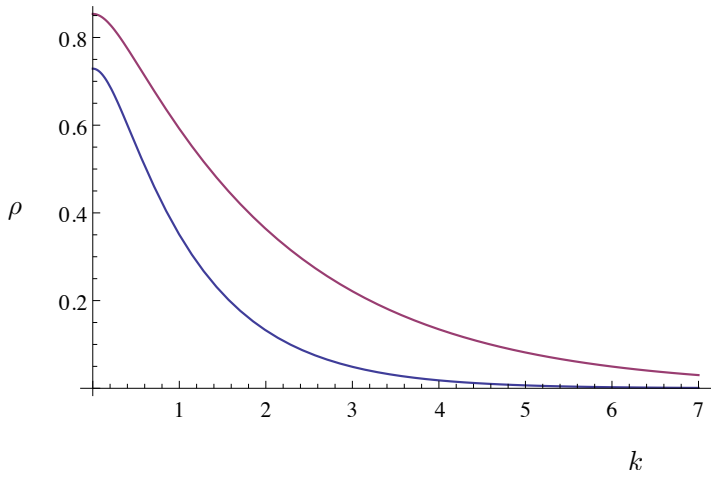

Figure 1.10: Convergence rate of the Schwarz method for $\eta = .1$, $\delta = 0.5$ (red curve) or $\delta = 1$ (blue curve).

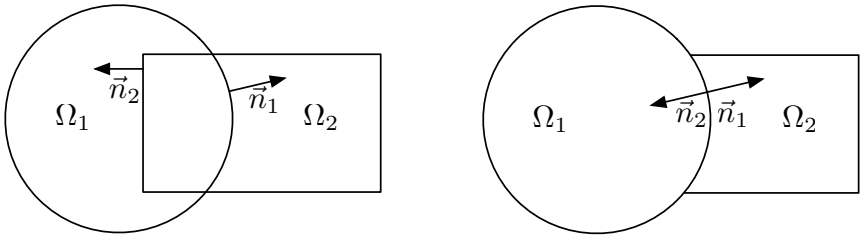

Figure 1.11: Outward normals for overlapping and non overlapping subdomains for P.L. Lions' algorithm.

conditions than the classical Dirichlet conditions at the interfaces between subdomains. The first more effective transmission conditions were first introduced by P.L. Lions' [19]. For elliptic problems, we have seen that Schwarz algorithms work only for overlapping domain decompositions and their performance in terms of iterations counts depends on the width of the overlap. The algorithm introduced by P.L. Lions [19] can be applied to both overlapping and non overlapping subdomains. It is based on improving Schwarz methods by replacing the Dirichlet interface conditions by Robin interface conditions.

Let α be a positive number, the modified algorithm reads

$$\begin{array}{rcll} -\Delta(u_1^{n+1}) & = & f & \text{in} \quad \Omega_1, \\ u_1^{n+1} & = & 0 & \text{on} \quad \partial\Omega_1 \cap \partial\Omega, \\ \left(\dfrac{\partial}{\partial \mathbf{n}_1} + \alpha\right)(u_1^{n+1}) & = & \left(\dfrac{\partial}{\partial \mathbf{n}_1} + \alpha\right)(u_2^{n}) & \text{on} \quad \partial\Omega_1 \cap \overline{\Omega_2}\,, \end{array} \tag{1.37}$$

and

$$\begin{array}{rcll} -\Delta(u_2^{n+1}) & = & f & \text{in} \quad \Omega_2, \\ u_2^{n+1} & = & 0 & \text{on} \quad \partial\Omega_2 \cap \partial\Omega \\ \left(\dfrac{\partial}{\partial \mathbf{n}_2} + \alpha\right)(u_2^{n+1}) & = & \left(\dfrac{\partial}{\partial \mathbf{n}_2} + \alpha\right)(u_1^{n}) & \text{on} \quad \partial\Omega_2 \cap \overline{\Omega_1} \end{array} \tag{1.38}$$

where $\mathbf{n}_1$ and $\mathbf{n}_2$ are the outward normals on the boundary of the subdomains, see Figure 1.11.

This algorithm was extended to Helmholtz problem by Després [10]. It is also possible to consider other interface conditions than Robin conditions and optimize their choice with respect to the convergence factor. All these ideas will be presented in detail in Chapter 2.

1.7 Schwarz methods using FreeFem++

The aim of this part is to illustrate numerically the previously defined Schwarz methods applied to second order elliptic boundary value problems (e.g Laplace equation and elasticity). In order to do this we will use the free finite element software FreeFem++ [14] developed at the Laboratoire Jacques-Louis Lions at Université Pierre et Marie Curie (Paris 6).

1.7.1 A very short introduction to FreeFem++

FreeFem++ allows a very simple and natural way to solve a great variety of variational problems by finite element type methods including Discontinuous Galerkin (DG) discretizations. It is also possible to have access to the underlying linear algebra such as the stiffness or mass matrices. In this

section we will provide only a minimal number of elements of this software, necessary for the understanding of the programs in the next section, see also `http://www.cmap.polytechnique.fr/spip.php?article239`. A very detailed documentation of FreeFem++ is available on the official website `http://www.freefem.org/ff++`, at the following address `http://www.freefem.org/ff++/ftp/freefem++doc.pdf` . The standard implementation includes tons of very useful examples that make a tutorial by themselves. It is also possible to use the integrated environment FreeFem++-cs [16] which provides an intuitive graphical interface to FreeFem++ users.

To start with, suppose we want to solve a very simple homogeneous Dirichlet boundary value problem for a Laplacian defined on a unit square $\Omega =]0,1[^2$:

$$\left\{ \begin{array}{rcll} -\Delta u & = & f & \text{in} \quad \Omega \\ u & = & 0 & \text{on} \quad \partial\Omega \end{array} \right. \tag{1.39}$$

The variational formulation of this problem reads:

$$\text{Find } u \in H_0^1(\Omega) := \{w \in H^1(\Omega) \ : \ w = 0 \text{ on } \partial\Omega\} \text{ such that}$$

$$\int_\Omega \nabla u.\nabla v dx - \int_\Omega f\, v\, dx = 0, \forall v \in H_0^1(\Omega)\,.$$

A feature of FreeFem++ is to penalize Dirichlet boundary conditions. The above variational formulation is first replaced by

$$\text{Find } u \in H^1(\Omega) \text{ such that}$$

$$\int_\Omega \nabla u.\nabla v dx - \int_\Omega f v\, dx = 0, \forall v \in H^1(\Omega)\,.$$

Then the finite element approximation leads to a system of the type

$$\sum_{j=1}^{M} A_{ij}u_j - F_j = 0,\ i = 1, ..., M,\ A_{ij} = \int_\Omega \nabla\phi_i.\nabla\phi_j dx, F_i = \int_\Omega \phi_i\, dx$$

where $(\phi_i)_{1\le i\le M}$ are the finite element functions. Note that the discretized system corresponds to a Neumann problem. Dirichlet conditions of the type $u = g$ are then implemented by penalty, namely by setting

$$A_{ii} = 10^{30},\ F_i = 10^{30} \cdot g_i$$

if i is a boundary degree of freedom. The penalty number 10^{30} is called TGV[1] and it is possible to change this value. The keyword `on` imposes the Dirichlet boundary condition through this penalty term.

[1] Très Grande Valeur (Terrifically Great Value) = Very big value in French

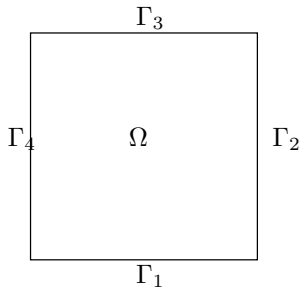

Figure 1.12: Numbering of square borders in FreeFem++

The following FreeFem++ script is solving this problem in a few lines. The text after `//` symbols are comments ignored by the FreeFem++ language. Each new variable must be declared with its type (here `int` designs integers).

```
3 // Number of mesh points in x and y directions
int Nbnoeuds=10;
```

Listing 1.1: `./FreefemCommon/survival.edp`

The function `square` returns a structured `mesh` of the square: the first two arguments are the number of mesh points according to x and y directions and the third one is a parametrization of Ω for x and y varying between 0 and 1 (here it is the identity). The sides of the square are labeled from 1 to 4 in trigonometrical sense (see Figure 1.12).

```
//Mesh definition
mesh Th=square(Nbnoeuds,Nbnoeuds,[x,y]);
```

Listing 1.2: `./FreefemCommon/survival.edp`

We define the function representing the right-hand side using the keyword `func`

```
// Functions of x and y
14 func f=x*y;
func g=1.;
```

Listing 1.3: `./FreefemCommon/survival.edp`

and the $P1$ finite element space `Vh` over the mesh `Th` using the keyword `fespace`

```
// Finite element space on the mesh Th
fespace Vh(Th,P1);
//uh and vh are of type Vh
Vh uh,vh;
```

Listing 1.4: `./FreefemCommon/survival.edp`

The functions u_h and v_h belong to the $P1$ finite element space V_h which is an approximation to $H^1(\Omega)$. Note here that if one wants to use $P2$ instead $P1$ finite elements, it is enough to replace `P1` by `P2` in the definition of `Vh`.

```
// variational problem definition
problem heat(uh,vh,solver=LU)=
        int2d(Th)(dx(uh)*dx(vh)+dy(uh)*dy(vh))
        -int2d(Th)(f*vh)
        +on(1,2,3,4,uh=0);
```

Listing 1.5: `./FreefemCommon/survival.edp`

The keyword `problem` allows the definition of a variational problem, here called `heat` which can be expressed mathematically as:
Find $u_h \in V_h$ such that

$$\int_\Omega \nabla u_h . \nabla v_h dx - \int_\Omega f v_h dx = 0, \forall v_h \in V_h \, .$$

Afterwards, for the Dirichlet boundary condition the penalization is imposed using TGV which is usually is equal to 10^{30}.

Note that keyword `problem` defines problem (1.39) without solving it. The parameter `solver` sets the method that will be used to solve the resulting linear system, here a Gauss factorization. In order to effectively solve the finite element problem, we need the command

```
//Solving the problem
heat;
// Plotting the result
plot(uh,wait=1);
```

Listing 1.6: `./FreefemCommon/survival.edp`

The FreeFem++ script can be saved with your favorite text editor (e.g. under the name `heat.edp`). In order to execute the script FreeFem++, it is enough to write the shell command `FreeFem++ heat.edp`. The result will be displayed in a graphic window.

One can easily modify the script in order to solve the same kind of problems but with mixed Neumann and Fourier boundary conditions such as

$$\begin{cases} -\Delta u + u &= f \text{ in } \Omega \\ \dfrac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_1 \\ u &= 0 \text{ on } \Gamma_2 \\ \frac{\partial u}{\partial n} + \alpha u &= g \text{ on } \Gamma_3 \cup \Gamma_4. \end{cases} \tag{1.40}$$

where f and g are arbitrary functions and α a positive real.
The new variational formulation consists in determining $u_h \in V_h$ such that

$$\int_\Omega \nabla u_h . \nabla v_h dx + \int_{\Gamma_3 \cup \Gamma_4} \alpha u_h v_h - \int_{\Gamma_3 \cup \Gamma_4} g v_h - \int_\Omega f v_h dx = 0,$$

for all $v_h \in V_h$. Here again the Dirichlet boundary condition will be penalized. The FreeFem++ definition of the problem reads:

```
// Changing boundary conditions to Neumann or Robin
real alpha =1.;
problem heatRobin(uh,vh)=
  int2d(Th)(dx(uh)*dx(vh)+dy(uh)*dy(vh))
  +int1d(Th,3,4)(alpha*uh*vh)
  -int1d(Th,3,4)(g*vh)
  -int2d(Th)(f*vh)
  +on(2,uh=0);
```

Listing 1.7: `./FreefemCommon/survival.edp`

In the variational formulation of (1.40) the extra boundary integral on $\Gamma_3 \cup \Gamma_4$ is defined by the keyword `int1d(Th,3,4)(function to integrate)`.

The keyword `varf` allows the definition of a variational formulation

```
// Using linear algebra package
varf varheatRobin(uh,vh)=
  int2d(Th)(dx(uh)*dx(vh)+dy(uh)*dy(vh))
  +int1d(Th,3,4)(alpha*uh*vh)
  -int1d(Th,3,4)(g*vh)
  -int2d(Th)(f*vh)
  +on(2,uh=0);
```

Listing 1.8: `./FreefemCommon/survival.edp`

If one wants to use some **linear algebra** package to solve the linear system resulting from the finite element discretisation, the program below shows how one can retrieve first the stiffness matrix and the vector associated to the right-hand side of the variational formulation. As a general rule, this procedure can be very useful if one wants to use other solvers such as domain decomposition methods. Here, the linear system is solved by UMFPACK [9].

```
// Retrieving the stiffness matrix
matrix Aglobal; // sparse matrix
Aglobal = varheatRobin(Vh,Vh,solver=UMFPACK); // stiffness matrix
                                              // UMFPACK direct solver
// Retrieving the right hand side
Vh rhsglobal;
rhsglobal[] = varheatRobin(0,Vh); //right hand side vector of d.o.f's
// Solving the problem by a sparse LU solver
uh[] = Aglobal^-1*rhsglobal[];
```

Listing 1.9: `./FreefemCommon/survival.edp`

Here `rhsglobal` is a finite element function and the associated vector of degrees of freedom is denoted by `rhsglobal[]`.

1.7.2 Setting the domain decomposition problem

According to the description of the Schwarz algorithms in the previous chapters, we need a certain number of data structures which will be built in the sequel. The file `data.edp` contains the declaration of these structures as well as the definition of the global problem to be solved.

```
load "metis" // mesh partitioner
load "medit" // OpenGL-based scientific visualization software
int nn=2,mm=2;       // number of the domains in each direction
int npart= nn*mm;      // total number of domains
int nloc = 20; // local no of dof per domain in one direction
bool withmetis = 1; // =1 (Metis decomp) =0 (uniform decomp)
int sizeovr = 1; // size of the geometric overlap between subdomains, algebraic
    ↪ overlap is 2 sizeovr+1
real allong = real(nn)/real(mm); // aspect ratio of the global domain
// Mesh of a rectangular domain
mesh Th=square(nn*nloc,mm*nloc,[x*allong,y]);// radial mesh
    ↪ [(1.+x*allong)*cos(pi*y),(1.+x*allong)*sin(pi*y)]);//
fespace Vh(Th,P1);
fespace Ph(Th,P0);
Ph part; // piecewise constant function
int[int] lpart(Ph.ndof); // giving the decomposition
// Domain decomposition data structures
mesh[int] aTh(npart); // sequence of subdomain meshes
matrix[int] Rih(npart); // local restriction operators
matrix[int] Dih(npart); // partition of unity operators
int[int] Ndeg(npart); // number of dof for each mesh
real[int] AreaThi(npart); // area of each subdomain
matrix[int] aA(npart),aN(npart); // local matrices
Vh[int] Z(npart); // coarse space, see Chapter 4
// Definition of the problem to solve
// Delta (u) = f, u = 1 on the global boundary
//int[int] chlab=[1,1 ,2,1 ,3,1 ,4,1 ];
//Th=change(Th,refe=chlab);   // all label borders are set to one
macro Grad(u) [dx(u),dy(u)] // EOM
func f = 1; // right hand side
func g = 0 ; // Dirichlet data
func kappa = 1.; // viscosity
func eta = 0;
Vh rhsglobal,uglob; // rhs and solution of the global problem
varf vaglobal(u,v) = int2d(Th)(eta*u*v+kappa*Grad(u)'*Grad(v))
                                    +on(1,u=g) + int2d(Th)(f*v);
matrix Aglobal;
// Iterative solver parameters
real tol=1e-6; // tolerance for the iterative method
int maxit=300; // maximum number of iterations
```

Listing 1.10: `./FreefemCommon/data.edp`

Afterwards we have to define a piecewise constant function `part` which takes integer values. The isovalues of this function implicitly defines a non overlapping partition of the domain. We have a coloring of the subdomains.

Suppose we want a decomposition of a rectangle Ω into `nn`×`mm` domains with approximately `nloc` points in one direction, or a more general partitioning method, using for example METIS [15] or SCOTCH [8]. In order to perform one of these decompositions, we make use of one of the

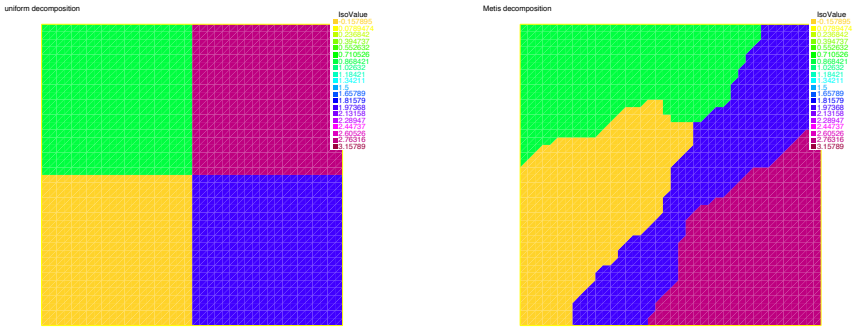

Figure 1.13: Uniform and Metis decomposition

routines `decompunif` or `decompMetis` defined in the script `decomp.idp` which will return a vector defined on the mesh, that can be recasted into the piecewise function `part` that we are looking for.

```
if (withmetis)
    {
      metisdual(lpart,Th,npart); // FreeFem++ interface to Metis
      for(int i=0;i<lpart.n;++i)
          part[][i]=lpart[i];
    }
  else
    {
    Ph xx=x,yy=y;
    part= int(xx/allong*nn)*mm + int(yy*mm);
    }
if (verbosity > 1)
   plot(part,wait=1,fill=1,value=1);
```

Listing 1.11: `./FreefemCommon/decomp.idp`

The isovalues of these two `part` functions correspond to respectively uniform or Metis non-overlapping decompositions as shown in Figure 1.13.

Using the function `part` defined as above as an argument into the routine `SubdomainsPartitionUnity`, we'll get as a result, for each subdomain labeled `i` the overlapping meshes `aTh[i]`:

```
func bool SubdomainsPartitionUnity(mesh & Th, real[int] & partdof, int sizeoverlaps, mesh[int] & aTh, matrix[int] & Rih, matrix[int] & Dih, int[int] & Ndeg, real[int] & AreaThi)
{
  int  npart=partdof.max+1;
  mesh Thi=Th; // freefem's trick, formal definition
  fespace Vhi(Thi,P1); // freefem's trick, formal definition
  Vhi[int] pun(npart); // local fem functions
  Vh sun=0, unssd=0;

  for(int ii=0;ii<npart;++ii)
    {

      int[int] arrayIntersection;
      int[int][int] restrictionIntersection(0);
      real[int] D;

       int numberIntersection = 0;


      mesh overlapName=Th;
          fespace VhGlobal(overlapName, P1);
      fespace PhGlobal(overlapName, P0);
      PhGlobal part;
      part[]=partdof;

          PhGlobal supp = abs(part - ii) < 0.1;
      Thi= trunc(overlapName, abs(part - ii) < 0.1, label = 10);
          VhGlobal suppSmooth;
          AddLayers(overlapName, supp[], sizeoverlaps * 2, suppSmooth[]);
          { mesh neighbors = trunc(overlapName, suppSmooth > 0.001 && (suppSmooth < 0.999));
              fespace Oh(neighbors, P0);
              Oh partOverlap = part;

              Unique(partOverlap[], arrayIntersection);

          }
          fespace Vhl(Thi, P1);
          Vhl[int] partitionIntersection(arrayIntersection.n);

          overlapName = trunc(overlapName, suppSmooth > 0.001);
          supp = supp;
          suppSmooth = suppSmooth;
          Thi = trunc(overlapName, suppSmooth> 0.501, label = 10);


      /*
      real oooo = 0.5001-0.5/sizeoverlaps;
          mesh borderName = trunc(overlapName, (suppSmooth >(sizeoverlaps-0.999 )/(2*sizeoverlaps)) && (suppSmooth < 0.501), label = 20);
          */


          Vhl khi = max(suppSmooth*2 - 1.0, 0.) ;
          if(usedARGV("-steep") != -1)
            khi = khi > 0.001 ? 1.0 : 0.0;
          Vhl sum = khi;
          VhGlobal phi = 0;
      real eps=int2d(overlapName)(1.);
          for(int i = 0; i < arrayIntersection.n; ++i) {
              PhGlobal suppPartition = abs(arrayIntersection[i] - part) < 0.1;
              AddLayers(overlapName, suppPartition[], sizeoverlaps, phi[]);
        if(usedARGV("-steep") != -1)
```

Listing 1.12: `./FreefemCommon/createPartition.idp`

Note that in the `CreatePartition.idp` script, the function `AddLayers` is called:

Listing 1.13: `./FreefemCommon/createPartition.idp`

These last two functions are tricky. The reader does not need to understand their behavior in order to use them. They are given here for sake of completeness. The restriction/interpolation operators `Rih[i]` from the local finite element space `Vh[i]` to the global one `Vh` and the diagonal local matrices `Dih[i]` are thus created.

Afterwards one needs to build the overlapping decomposition and the associated algebraic partition of unity, see equation (1.25). Program `testdecomp.edp` (see below) shows such an example by checking that the partition of unity is correct.

```
load "medit"

verbosity=2;
include "dataGENEO.edp"
include "decomp.idp"
include "createPartitionVec.idp"
SubdomainsPartitionUnityVec(Th,part[],sizeovr,aTh,Rih,Dih,Ndeg,AreaThi);
// check the partition of unity
Uh sum=0,fctone=1;
// for(int i=0; i < npart;i++)
// {
//  Uh localone;
// real[int] bi = Rih[i]*fctone[]; // restriction to the local domain
// real[int] di = Dih[i]*bi;
// localone[] = Rih[i]'*di;
// sum[] +=localone[] ;
// plot(localone,fill=1,value=1, dim=3,wait=1);
// }
// plot(sum,fill=1,value=1, dim=3,wait=1);
```

Listing 1.14: `./FreefemCommon/testdecomp.edp`

Suppose we want to do now the same thing in a **three-dimensional** case.

```
load "msh3"
func mesh3 Cube(int[int] & NN,real[int,int] &BB ,int[int,int] & L)
// basic functions to build regular mesh of a cube
// int[int] NN=[nx,ny,nz]; the number of seg in the 3 direction
// real [int,int] BB=[[xmin,xmax],[ymin,ymax],[zmin,zmax]]; bounding bax
// int [int,int] L=[[1,2],[3,4],[5,6]]; label of the 6 faces left,right, front, back, down, up
{
  // first build the 6 faces of the cube.
  real x0=BB(0,0),x1=BB(0,1);
  real y0=BB(1,0),y1=BB(1,1);
  real z0=BB(2,0),z1=BB(2,1);
  int nx=NN[0],ny=NN[1],nz=NN[2];
  mesh Thx = square(nx,ny,[x0+(x1-x0)*x,y0+(y1-y0)*y]);

  int[int] rup=[0,L(2,1)], rdown=[0,L(2,0)],
    rmid=[1,L(1,0), 2,L(0,1), 3, L(1,1), 4, L(0,0) ];
  mesh3 Th=buildlayers(Thx,nz, zbound=[z0,z1],
            labelmid=rmid, labelup = rup, labeldown = rdown);
  return Th;
}
```

Listing 1.15: `./FreefemCommon/cube.idp`

We would like to build a cube or a parallelepiped defined by calling the function `Cube` defined in the script `cube.idp` and then to split it into several domains. Again we need a certain number of data structures which will be declared in the file `data3d.edp`

```
load "metis"
load "medit"
int nn=2,mm=2,ll=2; // number of the domains in each direction
int npart= nn*mm*ll; // total number of domains
int nloc = 11; // local no of dof per domain in one direction
bool withmetis = 1; // =1 (Metis decomp) =0 (uniform decomp)
int sizeovr = 2; // size of the overlap
real allongx, allongz;
allongx = real(nn)/real(mm);
allongz = real(ll)/real(mm);
// Build the mesh
include "cube.idp"
int[int] NN=[nn*nloc,mm*nloc,ll*nloc];
real [int,int] BB=[[0,allongx],[0,1],[0,allongz]]; // bounding box
int [int,int] L=[[1,1],[1,1],[1,1]]; // the label of the 6 faces
mesh3 Th=Cube(NN,BB,L); // left,right,front, back, down, right
fespace Vh(Th,P1);
fespace Ph(Th,P0);
Ph part; // piecewise constant function
int[int] lpart(Ph.ndof); // giving the decomposition
// domain decomposition data structures
mesh3[int] aTh(npart); // sequence of ovr. meshes
matrix[int] Rih(npart); // local restriction operators
matrix[int] Dih(npart); // partition of unity operators
int[int] Ndeg(npart); // number of dof for each mesh
real[int] VolumeThi(npart); // volume of each subdomain
matrix[int] aA(npart); // local Dirichlet matrices
Vh[int] Z(npart); // coarse space
// Definition of the problem to solve
// Delta (u) = f, u = 1 on the global boundary
Vh intern;
intern = (x>0) && (x<allongx) && (y>0) && (y<1) && (z>0) && (z<allongz);
Vh bord = 1-intern;
macro Grad(u) [dx(u),dy(u),dz(u)] // EOM
func f = 1; // right hand side
func g = 1; // Dirichlet data
Vh rhsglobal,uglob; // rhs and solution of the global problem
varf vaglobal(u,v) = int3d(Th)(Grad(u)'*Grad(v))
                                  +on(1,u=g) + int3d(Th)(f*v);
matrix Aglobal;
// Iterative solver
real tol=1e-10; // tolerance for the iterative method
int maxit=200; // maximum number of iterations
```

Listing 1.16: `./FreefemCommon/data3d.edp`

Then we have to define a piecewise constant function `part` which takes integer values. The isovalues of this function implicitly defines a non overlapping partition of the domain. Suppose we want a decomposition of a rectangle Ω into `nn`×`mm` ×`ll` domains with approximately `nloc` points in one direction, or a more general partitioning method. We will make then use of one of

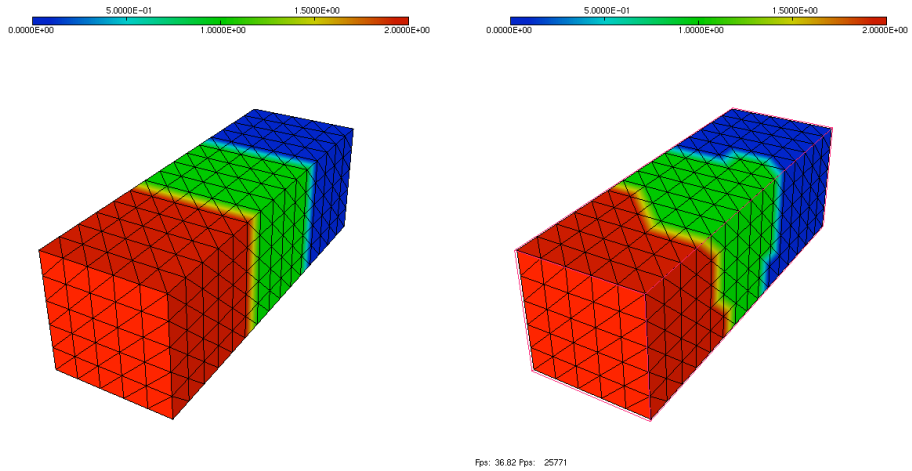

Figure 1.14: Uniform and Metis decomposition

the decomposition routines which will return a vector defined on the mesh, that can be recasted into the piecewise function `part` that we are looking for.

```
if (withmetis)
    {
      metisdual(lpart,Th,npart);
      for(int i=0;i<lpart.n;++i)
          part[][i]=lpart[i];
    }
  else
    {
    Ph xx=x,yy=y, zz=z;
    part= int(xx/allongx*nn)*mm*ll + int(zz/allongz*ll)*mm+int(y*mm);
    }
```

Listing 1.17: `./FreefemCommon/decomp3d.idp`

The isovalues of function `part` correspond to non-overlapping decompositions as shown in Figure 1.14.

Using the function `part` defined as above, function `SubdomainsPartitionUnity3` builds for each subdomain labeled `i` the overlapping meshes `aTh[i]`

```
func bool SubdomainsPartitionUnity3(mesh3 & Th, real[int] & partdof, int
    ↪ sizeoverlaps, mesh3[int] & aTh, matrix[int] & Rih, matrix[int] & Dih, int[int]
    ↪ & Ndeg, real[int] & VolumeThi)
{
  int  npart=partdof.max+1;
  mesh3 Thi=Th; // freefem's trick, formal definition
  fespace Vhi(Thi,P1); // freefem's trick, formal definition
  Vhi[int] pun(npart); // local fem functions
  Vh sun=0, unssd=0;
  Ph part;
  part[]=partdof;
  for(int i=0;i<npart;++i)
    {
      // boolean function 1 in the subdomain 0 elswhere
      Ph suppi= abs(part-i)<0.1;
      AddLayers3(Th,suppi[],sizeoverlaps,unssd[]); // overlapping partitions by
          ↪ adding layers
      Thi=aTh[i]=trunc(Th,suppi>0,label=10,split=1); // overlapping mesh,
          ↪ interfaces have label 10
      Rih[i]=interpolate(Vhi,Vh,inside=1); // Restriction operator : Vh -> Vhi
      pun[i][]=Rih[i]*unssd[];
      sun[] += Rih[i]'*pun[i][];
      Ndeg[i] = Vhi.ndof;
      VolumeThi[i] = int3d(Thi)(1.);
    }
  for(int i=0;i<npart;++i)
    {
      Thi=aTh[i];
      pun[i]= pun[i]/sun;
      Dih[i]=pun[i][];//diagonal matrix built from a vector
    }
  return true;
}
```

Listing 1.18: `./FreefemCommon/createPartition3d.idp`

by making use of the function `AddLayers3` in the `CreatePartition3d.idp`.

```
func bool AddLayers3(mesh3 & Th,real[int] &ssd,int n,real[int] &unssd)
{
  // build a continuous function uussd (P1) and modifies ssd :
  // IN: ssd in the caracteristics function on the input subdomain.
  // OUT: ssd is a boolean function, unssd is a smooth function
  // ssd = 1 if unssd >0; add n layer of element and unssd = 0 ouside of this layer
  Ph s;
  assert(ssd.n==Ph.ndof);
  assert(unssd.n==Vh.ndof);
  unssd=0;
  s[]= ssd;
  Vh u;
  varf vM(uu,v)=int3d(Th,qforder=1)(uu*v/volume);
  matrix M=vM(Ph,Vh);
  for(int i=0;i<n;++i)
    {
      u[]= M*s[];
      u = u>.1;
      unssd+= u[];
      s[]= M'*u[];
      s = s >0.1;
    }
  unssd /= (n);
  u[]=unssd;
  ssd=s[];
  return true;
}
```

Listing 1.19: `./FreefemCommon/createPartition3d.idp`

As in the 2D case, these last two functions are tricky. The reader does not need to understand their behavior in order to use them. They are given here for sake of completeness.

The restriction/interpolation operators `Rih[i]` from the local finite element space `Vh[i]` to the global one `Vh` and the diagonal local matrices `Dih[i]` are thus created. Afterwards one needs to build the overlapping decomposition and the associated algebraic partition of unity, see equation (1.25). Program `testdecomp3d.edp` shows such an example by checking that the partition of unity is correct.

```
include "data3d.edp"
include "decomp3d.idp"
include "createPartition3d.idp"
medit("part", Th, part, order = 1);
SubdomainsPartitionUnity3(Th,part[],sizeovr,aTh,Rih,Dih,Ndeg,VolumeThi);
// check the partition of unity
Vh sum=0,fctone=1;
for(int i=0; i < npart;i++)
{
  Vh localone;
    real[int] bi = Rih[i]*fctone[]; // restriction to the local domain
    real[int] di = Dih[i]*bi;
    localone[] = Rih[i]'*di;
    sum[] +=localone[] ;
    medit("loc",Th, localone, order = 1);
    medit("subdomains",aTh[i]);
}
medit("sum",Th, sum, order = 1);
```

Listing 1.20: `./FreefemCommon/testdecomp3d.edp`

1.7.3 Schwarz algorithms as solvers

We are now in a position to code Schwarz solvers. In program `schwarz-solver.edp` (see below) the RAS method (see eq. (1.29)) is implemented as a solver. First we need to split the domains into subdomains

```
verbosity=1;
include "../../FreefemCommon/dataGENEO.edp"
include "../../FreefemCommon/decomp.idp"
include "../../FreefemCommon/createPartition.idp"
SubdomainsPartitionUnity(Th,part[],sizeovr,aTh,Rih,Dih,Ndeg,AreaThi);
```

Listing 1.21: `./SCHWARZ/FreefemProgram/schwarz-solver.edp`

Then we need to define the global data from the variational formulation.

```
Aglobal = vaglobal(Vh,Vh,solver = UMFPACK); // global matrix
rhsglobal[] = vaglobal(0,Vh); // global rhs
uglob[] = Aglobal^-1*rhsglobal[];
plot(uglob,value=1,fill=1,wait=1,cmm="Solution by a direct method",dim=3);
```

Listing 1.22: `./SCHWARZ/FreefemProgram/schwarz-solver.edp`

Afterwords we build the local problem matrices

```
for(int i = 0;i<npart;++i)
  {
    cout << " Domain :" << i << "/" << npart << endl;
    matrix aT = Aglobal*Rih[i]';
    aA[i] = Rih[i]*aT;
    set(aA[i],solver = UMFPACK);// direct solvers
  }
```

Listing 1.23: `./SCHWARZ/FreefemProgram/schwarz-solver.edp`

and finally the Schwarz iteration

```
ofstream filei("Conv.m");
Vh un = 0; // initial guess
Vh rn = rhsglobal;
for(int iter = 0;iter<maxit;++iter)
  {
    real err = 0, res;
    Vh er = 0;
    for(int i = 0;i<npart;++i)
      {
        real[int] bi = Rih[i]*rn[]; // restriction to the local domain
        real[int] ui = aA[i]^-1* bi; // local solve
        bi = Dih[i]*ui;
        // bi = ui; // uncomment this line to test the ASM method as a solver
        er[] += Rih[i]'*bi;
      }
    un[] += er[]; // build new iterate
    rn[] = Aglobal*un[]; // computes global residual
    rn[] = rn[] - rhsglobal[];
    rn[] *= -1;
    err = sqrt(er[]'*er[]);
    res = sqrt(rn[]'*rn[]);
    cout << "Iteration: " << iter << " Correction = " << err << " Residual = "
        ↪ << res << endl;
    plot(un,wait=1,value=1,fill=1,dim=3,cmm="Approximate solution at step " +
        ↪ iter);
    int j = iter+1;
    // Store the error and the residual in Matlab/Scilab/Octave form
    filei << "Convhist("+j+",:)=[" << err << " " << res <<"];" << endl;
    if(err < tol) break;
  }
plot(un,wait=1,value=1,fill=1,dim=3,cmm="Final solution");
```

Listing 1.24: `./SCHWARZ/FreefemProgram/schwarz-solver.edp`

The convergence history of the algorithm is stored in a Matlab file (also compatible with Scilab or Octave) `Conv.m`, under the form of a two-column matrix containing the error evolution as well as the residual one.

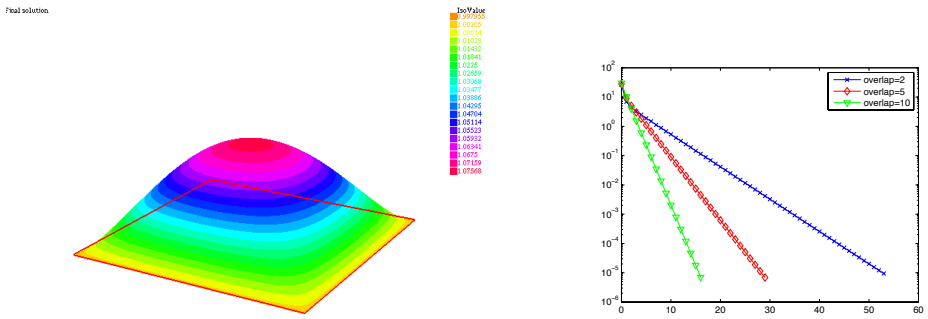

Figure 1.15: Solution and RAS convergence as a solver for different overlaps

The result of tracing the evolution of the error is shown in Figure 1.15 where one can see the convergence history of the RAS solver for different values of the overlapping parameter.

Remark 1.7.1 *Previous tests have shown a very easy use of the RAS iterative algorithm and some straightforward conclusions from this.*

- *The convergence of RAS, not very fast even in a simple configuration of 4 subdomains, improves when the overlap is getting bigger.*
- *Note that it is very easy to test the ASM method, see eq. (1.30), when used as a solver. It is sufficient to uncomment the line* `bi = ui;`.
- *Running the program shows that the ASM does not converge. For this reason, the ASM method is always used a preconditioner for a Krylov method such as CG, GMRES or BiCGSTAB, see chapter 3.*
- *In the the three-dimensional case the only part that changes is the decomposition into subdomains. The other parts of the algorithm are identical.*

```
include "../../FreefemCommon/data3d.edp"
include "../../FreefemCommon/decomp3d.idp"
include "../../FreefemCommon/createPartition3d.idp"
SubdomainsPartitionUnity3(Th,part[],sizeovr,aTh,Rih,Dih,Ndeg,VolumeThi);
```

Listing 1.25: `./SCHWARZ/FreefemProgram/schwarz-solver3d.edp`

1.7.4 Systems of PDEs: the example of linear elasticity

Suppose we want to solve now another kind of problem, such a linear elasticity. A few changes will be necessary.

```
load "metis"
load "medit"
int nn=3,mm=3;       // number of the domains in each direction
int npart= nn*mm;      // total number of domains
int nloc = 20; // local no of dof per domain in one direction
bool withmetis = 1; // =1 (Metis decomp) =0 (uniform decomp)
int sizeovr = 2; // size of the overlap
real allong = real(nn)/real(mm); // aspect ratio of the global domain
func E = 2*10^11; // Young modulus ans Poisson ratio
func sigma = 0.3;
func lambda = E*sigma/((1+sigma)*(1-2*sigma)); // Lame coefficients
func mu = E/(2*(1+sigma));
real sqrt2=sqrt(2.);
func eta = 1.0e-6;
// Mesh of a rectangular domain
mesh Th=square(nn*nloc,mm*nloc,[x*allong,y]);
fespace Vh(Th,[P1,P1]); // vector fem space
fespace Uh(Th,P1); // scalar fem space
fespace Ph(Th,P0);
Ph part; // piecewise constant function
int[int] lpart(Ph.ndof); // giving the decomposition
// Domain decomposition data structures
mesh[int] aTh(npart); // sequence of ovr. meshes
matrix[int] Rih(npart); // local restriction operators
matrix[int] Dih(npart); // partition of unity operators
int[int] Ndeg(npart); // number of dof for each mesh
real[int] AreaThi(npart); // area of each subdomain
matrix[int] aA(npart); // local Dirichlet matrices
// Definition of the problem to solve
int[int] chlab=[1,11 ,2,2 ,3,33 ,4,1 ]; //Dirichlet conditions for label = 1
Th=change(Th,refe=chlab);
macro Grad(u) [dx(u),dy(u)] // EOM
macro epsilon(u,v) [dx(u),dy(v),(dy(u)+dx(v))/sqrt2] // EOM
macro div(u,v) ( dx(u)+dy(v) ) // EOM
func uboundary = (0.25 - (y-0.5)^2);
varf vaBC([u,v],[uu,vv]) = on(1, u = uboundary, v=0) + on(11, u = 0, v=0) + ↵
    ↳ on(33, u=0,v=0);
// global problem
Vh [rhsglobal,rrhsglobal], [uglob,uuglob];
macro Elasticity(u,v,uu,vv) eta*(u*uu+v*vv) + ↵
    ↳ lambda*(div(u,v)*div(uu,vv))+2.*mu*( epsilon(u,v)'*epsilon(uu,vv) ) // ↵
    ↳ EOM
varf vaglobal([u,v],[uu,vv]) = int2d(Th)(Elasticity(u,v,uu,vv)) + vaBC; // ↵
    ↳ on(1,u=uboundary,v=0)
matrix Aglobal;
// Iterative solver parameters
real tol=1e-6; // tolerance for the iterative method
int maxit=200; // maximum number of iterations
```

Listing 1.26: `./FreefemCommon/dataElast.edp`

contains now the declarations and data. The definition of the partition is done like before using `decomp.idp`. The `SubdomainsPartitionUnityVec` is the vector adaptation of `SubdomainsPartitionUnity` and will provide the same type of result

```
func bool SubdomainsPartitionUnityVec(mesh & Th, real[int] & partdof, int
    ↪ sizeoverlaps, mesh[int] & aTh, matrix[int] & Rih, matrix[int] & Dih, int[int]
    ↪ & Ndeg, real[int] & AreaThi)
{
  int  npart=partdof.max+1;
  mesh Thi=Th; // freefem's trick, formal definition
  fespace Vhi(Thi,[P1,P1]); // freefem's trick, formal definition
  Vhi[int] [pun,ppun](npart); // local fem functions
  Vh [unssd,uunssd], [sun,ssun]=[0,0];
  Uh Ussd = 0;
  Ph part;
  int[int] U2Vc=[0,1]; // no component change
  part[]=partdof;
  for(int i=0;i<npart;++i)
    {
      Ph suppi= abs(part-i)<0.1; // boolean 1 in the subdomain 0 elswhere
      AddLayers(Th,suppi[],sizeoverlaps,Ussd[]); // ovr partitions by adding layers
      [unssd,uunssd] =[Ussd,Ussd];
      Thi=aTh[i]=trunc(Th,suppi>0,label=10,split=1); // ovr mesh interfaces label
          ↪ 10
      Rih[i]=interpolate(Vhi,Vh,inside=1,U2Vc=U2Vc); // Restriction operator :
          ↪ Vh -> Vhi
      pun[i][]=Rih[i]*unssd[];
      sun[] += Rih[i]'*pun[i][];
      Ndeg[i] = Vhi.ndof;
      AreaThi[i] = int2d(Thi)(1.);
   }
  for(int i=0;i<npart;++i)
    {
      Thi=aTh[i];
      [pun[i],ppun[i]] = [pun[i]/sun, ppun[i]/sun];
      Dih[i]=pun[i][]; //diagonal matrix built from a vector
    }
  return true;
}
```

Listing 1.27: `./FreefemCommon/createPartitionVec.idp`

Note that in the `CreatePartitionVec.idp` script, the function `AddLayers` is called:

```
func bool AddLayers(mesh & Th,real[int] &ssd,int n,real[int] &unssd)
{
  // build a continuous function uussd (P1) and modifies ssd :
  // IN: ssd in the caracteristics function on the input subdomain.
  // OUT: ssd is a boolean function, unssd is a smooth function
  // ssd = 1 if unssd >0; add n layer of element and unssd = 0 ouside of this layer
  Ph s;
  Uh u;
  assert(ssd.n==Ph.ndof);
  assert(unssd.n==Uh.ndof);
  unssd=0;
  s[]= ssd;
  varf vM(uu,v)=int2d(Th,qforder=1)(uu*v/area);
  matrix M=vM(Ph,Uh);
  for(int i=0;i<n;++i)
    {
      u[]= M*s[];
      u = u>.1;
      unssd+= u[];
      s[]= M'*u[];
      s = s >0.1;
    }
  unssd /= (n);
  u[]=unssd;
  ssd=s[];
  return true;
}
```

Listing 1.28: `./FreefemCommon/createPartitionVec.idp`

The restriction/interpolation operators `Rih[i]` from the local finite element space `Vh[i]` to the global one `Vh` and the diagonal local matrices `Dih[i]` are thus created.

We are now in a position to code Schwarz solvers. In program `schwarz-solver-elast.edp` (see below) the RAS method (see eq. (1.29)) is implemented as a solver following the same guidelines as in the case of the Laplace equation. First we need to split the domains into subdomains

```
include "../../FreefemCommon/dataElast.edp"
include "../../FreefemCommon/decomp.idp"
include "../../FreefemCommon/createPartitionVec.idp"
SubdomainsPartitionUnityVec(Th,part[],sizeovr,aTh,Rih,Dih,Ndeg,AreaThi);
```

Listing 1.29: `./SCHWARZ/FreefemProgram/schwarz-solver-elast.edp`

Then we need to define the global data from the variational formulation.

```
Aglobal = vaglobal(Vh,Vh,solver = UMFPACK); // global matrix
rhsglobal[] = vaglobal(0,Vh); // global rhs
uglob[] = Aglobal^-1*rhsglobal[];
real coeff2 = 1;
mesh Thmv=movemesh(Th,[x+coeff2*uglob,y+coeff2*uuglob]);
medit("Thmv", Thmv);
medit("uex", Th, uglob, Th, uuglob, order=1);
```

Listing 1.30: `./SCHWARZ/FreefemProgram/schwarz-solver-elast.edp`

Afterwords, the local problem matrices are built in the same way as before and finally the Schwarz iteration

```
ofstream filei("Conv.m");
Vh [un,uun] = [0,0]; // initial guess
Vh [rn,rrn] = [rhsglobal,rrhsglobal];
for(int iter = 0;iter<maxit;++iter)
  {
    real err = 0, res;
    Vh [er,eer] = [0,0];
    for(int i = 0;i<npart;++i)
      {
        real[int] bi = Rih[i]*rn[]; // restriction to the local domain
        real[int] ui = aA[i] ^-1* bi; // local solve
        bi = Dih[i]*ui;
        // bi = ui; // uncomment this line to test the ASM method as a solver
        er[] += Rih[i]'*bi;
      }
    un[] += er[]; // build new iterate
    rn[] = Aglobal*un[]; // computes global residual
    rn[] = rn[] - rhsglobal[];
    rn[] *= -1;
    err = sqrt(er[]'*er[]);
    res = sqrt(rn[]'*rn[]);
    cout << "Iteration: " << iter << " Correction = " << err << " Residual = "
        ↪ << res << endl;
    int j = iter+1;
    // Store the error and the residual in Matlab/Scilab/Octave form
    filei << "Convhist("+j+",:)=[" << err << " " << res <<"];" << endl;
    if(res < tol) break;
  }
mesh Thm=movemesh(Th,[x+coeff2*un,y+coeff2*uun]);
medit("Thm", Thm);
medit("uh", Th, un, Th, uun, order=1);
```

Listing 1.31: `./SCHWARZ/FreefemProgram/schwarz-solver-elast.edp`